Assessing the effect of source code characteristics on changeability

Thesis

ASSESSING THE EFFECT OF SOURCE CODE CHARACTERISTICS ON CHANGEABILITY

Angela C. Lozano Rodríguez

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University
United Kingdom

June 2009

Abstract

Maintenance is the phase of the software lifecycle that comprises any modification after the delivery of an application. Modifications during this phase include correcting faults, improving internal attributes, as well as adapting the application to different environments. As application knowledge and architectural integrity degrade over time, so does the facility with which changes to the application are introduced. Thus, eliminating source code that presents characteristics that hamper maintenance becomes necessary if the application is to evolve. We group these characteristics under the term Source Code Issues. Even though there is support for detecting Source Code Issues, the extent of their harmfulness for maintenance remains unknown.

One of the most studied Source Code Issue is cloning. Clones are duplicated code, usually created as programmers copy, paste, and customize existing source code. However, there is no agreement on the harmfulness of clones.

This thesis proposes and follows a novel methodology to assess the effect of clones on the changeability of methods. Changeability is the ease with which a source code entity is modified. It is assessed through metrics calculated from the history of changes of the methods. The impact of clones on the changeability of methods is measured by comparing the metrics of methods that contain clones to those that do not. Source code characteristics are then tested to establish whether they are endemic of methods whose changeability decay increase when cloned.

We analyzed five Java open source applications and found that cloning indeed affects the changeability of methods; and that, in general, cloned methods have a poorer changeability than methods not cloned. To be more precise, we found that methods that are cloned a fraction of their lifetime, (which represent roughly half of the clones), have significant changeability decay, whereas methods that were cloned their whole lifetime had a similar changeability to methods that were never cloned. From these findings, we conclude that if maintenance tasks focus on tackling clones indiscriminately, half of the resources will be wasted on eliminating harmless clones. We also found that most of the changes in cloned methods occur in their cloned fragments. This confirms that the increase of changeability decay on cloned methods is due to their cloned fragments. We found that, although some characteristics are correlated with changeability decay, none of them could distinguish cloned methods with high changeability decay on their own. A combination of clone and method characteristics is necessary in order to

identify harmful clones.

In addition to findings on the harmfulness of cloning, this thesis contributes a methodology that can be applied to assess the harmfulness of other Source Code Issues.

The contributions of this thesis are twofold. First, the findings answer the question about the harmfulness of clones on changeability by showing that cloned methods are more likely to change, and that some cloned methods have significantly higher changeability decay when cloned. Furthermore, it offers a characterization of such harmful clones. Second, the methodology provides a guide to analyze the effect of Source Code Characteristics in changeability; and therefore, can be adapted for other Source Code Issues.

Author's Declaration

Several parts of thesis have been published (in different form) in the following papers

- Lozano, A., Wermelinger, M.: Assessing the effect of clones on changeability. In Proc. of the 24th Int'l Conf. on Software Maintenance (ICSM): IEEE Computer Society, 2008

- Lozano, A., Wermelinger, M., Nuseibeh, B.: Evaluating the relation between changeability decay and the characteristics of clones and methods. In Proc. of the Int'l Workshop on Software Evolution (Evol): ERCIM, 2008

- Lozano, A., Wermelinger, M., Nuseibeh, B.: Evaluating the harmfulness of cloning: a change based experiment. In Proc. of the 4th Int'l Workshop on Mining Software Repositories (MSR): IEEE Computer Society, 2007

- Lozano, A.: A Methodology to Assess the Impact of Source Code Flaws in Changeability, and its Application to Clones (Doctoral Symposium). In Proc. of the 24th Int'l Conf. on Software Maintenance (ICSM): IEEE Computer Society, 2008

- Lozano, A., Wermelinger, M., Nuseibeh, B.: Assessing the impact of bad smells using historical information (Position paper). In Proc. of the 9th Int'l Workshop On Principles of Software Evolution (IWPSE): IEEE Computer Society, 2007

- Lozano, A., Wermelinger, M., Nuseibeh, B.: Degradation archaeology: studying software flaws' evolution (Position paper). In Proc. of the Int'l Workshop on Software Evolution (Evol): ERCIM, 2006

None of this material has been submitted previously for any degree, at any institution. The supervisors of this work have guided the definition of research questions, and methodological choices. Nevertheless, all the work presented in this thesis describes original contributions of the author.

Acknowledgements

This thesis was completed thanks to the Open University, to their financial support, and their trust in my potential to achieve it. Especially to the academic and personal support of my supervisors: Dr. Michel Wermelinger, and Prof. Bashar Nuseibeh. I owe much to Michel for his open-mindedness, strong commitment, and support along these four years. His enthusiasm, skeptical attitude, and patient guidance shaped my research interests, and helped me to build my argument. Thanks to Bashar, I learnt the value of old publications, the importance of enjoying your work, and the craftsmanship and detail that arguments require. His continuous encouragement to submit my work to the toughest venues gave me confidence whenever I had lost my own.

Apart from my supervisors, I owe to Prof. Marian Petre my general training as researcher. I am very grateful with Marian for teaching me what research is about, the importance of evidence, rigor, and argument. Her sharing of not only the fundamentals but also of the details, the anecdotes, and the warnings has been a constant reference in my work. I am sure that her teachings will remain the core of my professional career. However, my gratitude towards Marian does not cover only the professional side, but also the personal side. I owe many thanks to Marian for opening to me the doors of her house, for sharing with me as another family member, but specially, for her disposition to hear my worries and offering me her advice. My time in Milton Keynes would have not been the same without her.

Above all personal and professional support, I have to thank Michel, Bashar, and Marian for their example. Regardless of their excellent reputation as researchers, they are humble and approachable. This attitude gave me the best lesson I learnt from this journey: people is what matters the most.

There are many people in the department that have welcomed me, and made me feel as a valuable member of the department, maybe too many to mention them all. I feel honored to have had the chance of discussing my work with Yijun Yu and Juan Fernandez Ramil, and I feel grateful with them for treating me as a colleague. I was also lucky of having Janet van der Linden as third party monitor. Janet has been a confident of my worries all along the PhD, caring for what I cared the most, and communicating my concerns to those that could make a

difference.

Apart from the academic staff in the department, I have to thank several people for solving my complicated administrative/bureaucratic problems. In particular, to Paula Piggott and Kathryn Reeves of the Research School, for understanding all the particularities I had to dealt with to get visas, and the issues that creates being a full time student that is not living in the UK. And to Kerry Tucker, Debbie Briggs, Gemma Chapman of the Computing Department for helping me to deal with procedures like the progress monitor report, and the claim expenses system.

I also have to thank people outside the Computing Department. Thanks to Prof. Laurence Duchien and to Prof. Lionel Seinturier for hosting me in their research group (ADAM-INRIA-Lille), but specially for adopting me as another researcher in the group for the last year of my PhD. Thanks also to Dr. Dirk Deridder and Prof. Viviane Jonckers for inviting me to their research group (SSEL-VUB-Brussels) while I wrote my thesis. The chance of having a research community and an office to work while I was living outside of the UK has been a key factor in finishing my thesis.

I am very grateful with those people that have diligently assessed my work. I have to thank to Dr. Jon Hall and Dr. Janet van der Linden for providing me with an experience of a viva. Especially to John for his detailed reading of this dissertation. Their questions, feedback, and suggestions gave me a reality check of what it takes to have a successful viva. Thanks also to Prof. Helen Sharp and Dr. Jim Buckley for accepting being my thesis examiners. I know that reading a PhD thesis is a time-consuming process, and that you have busy schedules. I really appreciate their view of my work as well comments to improve it.

I owe also to fellow students for their expert opinion. To Andrea Capiluppi, and Israel Herraiz for offering me a safe and comfortable space of discussion. Their gentle criticism helped me to reflect on the focus that my work would take. It was a great benefit to hear their opinion and pointers at the beginning of my research project. To Karim Anaya-Izquierdo, and David Jenkinson for teaching me about the power and subtleties of statistical hypothesis testing; but also, for always being open to questions, for the dummy explanations, and the great consideration with my novice status in the area. To Charles and Debra Haley, and to Geke van Dijck for the selfless advice on everything I would ever asked them, they saved me a steep learning curve on tools that go from Word, to EndNote and statistical packages.

Before thanking my family, I have to thank the people that became my friends along this

journey. To Ekaterini Tzanidou for offering me grounded advice without making me feel disheartened, for offering me her support, but specially for giving me her trust. Thanks to Kate, I started to feel in the UK a little bit at home. To Ann Abrahams for offering me her friendship, for being always on my side, and giving me advice ranging from technical details to personal counseling. Ann made of the OU a special place. I also want to thank Frederick Loiret and Ales Plesk for all those nice discussions around the lunch table in Lille, for being more than colleagues, and for valuing my point of view in technical, political, research, and even personal matters. To Andres Yie, Adriana Manrique for being there since our arrival, for making of Brussels a more welcoming place, and for reminding us of the Colombian warmth without making us feel nostalgia from home.

I have to thank also those directly responsible for my success, my family. I have to thank my parents, Esperanza and Luis Carlos, for teaching me to be strong even if I feel alone, to fight for what I believe in, and to value whatever other people can teach me. I have to thank my mom for teaching me the love for culture, the fascination for difference, for science, for the value of arguments. I have to thank my dad, for teaching me the importance of hard work, of being honest and truthful regardless everything else. I also have to thank my siblings Andrea Paola and Luis Carlos for their own achievements while I was doing the PhD because they encouraged me to keep being an example for them; but also, for their unconditional support, and for their messages of encouragement whenever I was downhearted. Thanks to the sacrifices of my family, I was able to get a first class education; I hope I can repay them with more than pride. Although not by blood, my in laws are very much my family. I have to thank my parents-in-law for their vote of confidence in our life-plan, for their reassurance, and their emotional, and financial support whenever we needed them.

Finally, the person that I have to thank the most: Carlos. Without his unconditionally and thorough support this work would have not been done. It is great to have a life partner that is also an intellectual partner! He believed in me and on my work, he encouraged me and inspired me, he helped me to see it from other angles, he helped me to become more objective, he read my documents and carefully reviewed all of them, he took care with life-logistics while I was in full-time-write mode, he suggested venues, authors, topics. He was always there, regardless of time, place, mood, problems...Thanks a lot! :-).

Table of Contents

List of Figures

List of Tables

List of Equations

To Carlos

Chapter 1. Introduction

Software applications are critical resources for organizations. Therefore, the longer an application supports the needs of the organization, the more successful it is. However, unless specific actions are taken to improve the longevity of an application, an increase in the difficulty of keeping it fulfilling user's requirements is inevitable. There are two explanations for this phenomenon: over time, knowledge of the application gets lost, and the application becomes over-complex. One alternative suggested to tackle these issues is to keep a design intuitive and self documented, so that it remains easy to understand, and, therefore, easy to change. Nevertheless, it is not clear to what extent the positive effects of a good design also apply to source code.

This chapter explains the need for understanding the effect of source code characteristics on maintenance, introduces the research questions that motivate this work, presents the proposed approach to assess the effect of such source code characteristics on maintenance, and finally, summarizes the rest of this thesis.

1.1 Motivation: structure and maintenance

This section makes a case for the importance of knowing which source code characteristics affect longevity of an application. This section defines maintenance, its importance, the difficulties of maintaining an application, and how some of these difficulties can be dealt by using an appropriate structure for the application. Finally, this section argues that not only the design, but also the source code may help to facilitate maintenance.

This thesis uses the terms software maintenance and software evolution in the same way as [Bennett '00]. Software maintenance is the phase of the software lifecycle when the system is modified after its first delivery [Bennett '00]. The attributes of an application that bear on the effort needed to make modifications is called maintainability [Sanders '95]. Software evolution is the first sub-phase of maintenance, in which the application undergoes major changes like the addition of new functionality or adaptation to new environments [Bennett '00]. In order to refer

to the life of the application (called evolution by Lehman and Belady [Lehman '85]) we will use the term software history. Finally, to refer to the way in which a property changes over time we call it the evolution of the property.

1.1.1 On the importance of software maintenance

Software systems are expected to have a long life due to their cost. Successful applications fulfill user's requirements. In order to keep fulfilling the requirements of its users, the application requires modifications to fix bugs, to implement requirements that were not foreseen, to enhance functionality for a changing environment, or to implement requirements that ripened with the use of the system [Lehman '85].

Software maintenance is the longest and most expensive stage of the software lifecycle; therefore, software maintenance is a critical stage. Besides, the better an application is evolved; that is, the better major changes are integrated to keep the design intuitive and self-documented, the longer its maintenance phase will be. In consequence, software evolution is a critical phase of software maintenance. In fact, some studies have found that more than half of professional programmers' time [Lientz '81; Singer '98] is consumed on maintenance tasks. Furthermore, more than 40% of maintenance effort is spent on enhancements and extensions [Lientz '81] , that is, on the evolution phase .

Therefore, if concrete properties that reduce or increase maintenance in the long term are identified, the costs of software development can be significantly reduced. Besides, such properties could provide insights in the research of software quality.

1.1.2 Difficulties of achieving long term maintenance

The difficulty of performing maintenance tasks increases with the age of application, and with the amount of maintenance tasks performed on the application. This section discusses the relation between the age of an application, and the difficulty to perform changes on it.

1.1.2.1 Key elements to achieve long term maintenance

The evolution phase is facilitated when there is software architecture integrity and software team knowledge, as “they allow the team to make substantial changes without damaging the architectural integrity” [Bennett '00]. A less coherent architecture requires more extensive knowledge in order to evolve it, and a lack of knowledge results in a faster deterioration of the

architecture [Bennett '00]. However, neither the integrity of the architecture nor the knowledge about the system remains as the system evolves.

1.1.2.2 Difficulties of maintaining the system's knowledge

The loss of system's knowledge is difficult to avoid for two reasons: first because documentation is inaccurate, and second because the knowledge of programmers about the application is not kept.

From early studies, documentation has been found to be one of the top maintenance problems [Lientz '81]. Inadequate documentation affects maintenance. Basili and Perricone [Basili '84] found that misunderstanding of a module's specifications or requirements, and mistaken assumptions constituted the main factors on provoking defects when modifying existing code.

The recording of design rationale is a complex task. Therefore, documentation tends to have several issues:, it is poorly organized [Parnas '94], imprecise [Parnas '94], incomplete [Parnas '94], it has uneven coverage and in some cases it is incoherent. Even if the documentation is used, usually it is not trusted [Singer '98], so it is likely that the only or most reliable source of information about an application is its source code.

Besides, as time passes, programmers leave the project taking with them the domain and system knowledge they have acquired. When maintainers do not know the original design they might have difficulties of agreeing on the system's components and their relations, or grasping how the whole application works [Parnas '94]. The lack of knowledge may be more frequent on organizations that have separate developer and maintenance teams, or that have a high turnover in the maintenance team.

1.1.2.3 Difficulties of maintaining the architecture integrity

The loss of architecture integrity seem to be an inevitably result of maintenance. Such loss of architecture integrity makes future maintenance more difficult. Evolving an application requires introducing changes without introducing defects. Nevertheless, accidental defects might be introduced every time the application is changed. Therefore, the cost of a change can grow exponentially with respect to the system's age [Lehman '85]. Besides, if changes do not comply with the original design contracts, the design degrades [Parnas '94]. An eroded design implies that neither original programmers nor maintainers have enough knowledge about the system as a whole [Parnas '94]. Therefore, the system becomes expensive to update because "changes take

longer and are more likely to introduce new bugs" [Parnas '94]. Given that these events are directly related with the applications' age, it is said that the software is aging[1] whenever the application becomes incapable to accommodate changing needs[Parnas '94], i.e. when it can only accommodate minor changes. There is some empirical evidence that supports theory of degradation in the architecture. For instance, changes lose locality over time, this may indicate that the modularization degrades, as it is not hiding changes behind abstractions [Eick '01]. Furthermore, the degree of connectivity inside and across software components, and inside and across abstraction layers also increases over time [Bianchi '01]. Moreover, the minimal path of objects needed to traverse to access a second object increases over time [Burd '99].

Keeping the architecture integrity over time is known as **anti-regressive work** [Lehman '85]. The goal of anti-regressive work is to keep the design intuitive and self documented, so that it remains easy to understand and therefore easy to change. Anti-regressive work is to software as waste collection, recycling, pest control, research in alternative energy sources, etc. is to cities. [Lehman '85]. This means that, anti-regressive work avoids degradation but it does not add, adapt or enhance functionality. It does not improve the perception of the final user, but it improves the perception of the developer in charge of the application i.e. it improves the maintainability [Lehman '85]. In that sense, although anti-regressive work is not an urgent task, it is very important because anti-regressive work affects the ease of introducing changes and, as a result, the life expectancy of the application. Therefore, it is necessary to allocate time for anti-regressive work, and use that time efficiently.

Summarizing, software maintenance is the longest and most expensive stage in the software lifecycle. Software maintenance requires changes in the application to keep supporting users' needs. However, achieving those changes in the long term is difficult because the original developers may not work anymore in the application, and because the introduction of changes increases the complexity of the application's structure making the introduction of future changes difficult. In order to facilitate future changes, it is necessary to do anti-regressive work to keep the complexity stable, and the design understandable. The following section discusses in more

[1] Software aging has been used in two ways: (i) Systems incapable of meeting recent needs of its users, due to structural degradation. Such structural degradation is usually due to modifications that did not comply with the original design [Parnas '94]. (ii)Systems that run for long enough that some performance problems emerge (e.g. due to inaccessible memory because of lost pointers) until the system halts and needs to be re-started [Grottke '06].

detail the relation between the structure of the application and its maintainability.

1.1.3 A good design facilitates maintenance

A good design reflects the domain concepts and provides abstractions to separate the concerns among those concepts. Such characteristics are highly desirable for maintenance tasks because bugs are introduced by lack of awareness of the parts involved in executing a requirement [Coleman '94], and because maintenance time is spent on understanding the application to decide which parts to change [Singer '98].

Most models proposed to measure maintainability take into account the quality of the application's structure. Figure 1-1 shows several models proposed to assess maintenance; all references to the structure quality are shown in italics. Notice that all models identify structural properties as key elements of maintenance.

In fact, there are empirical indicators to believe that software structure affects software quality. For instance, several studies have found that structural metrics are good indicators of software defects [Basili '96; Ferneley '99; Subramanyam '03]. Furthermore, other studies have found that structural metrics are highly correlated with maintenance effort [Coleman '94; Harrison '98; Fioravanti '01; Bandi '03; Dagpinar '03].

A software application has several levels of granularity: architecture, design, and source code. Therefore, the structure of software may facilitate software maintenance. Nevertheless, it is unclear whether all levels of granularity of software structure improve software maintenance or if different levels of granularity have different impacts. In particular, it is not clear if the source code must satisfy any requirements to achieve a maintainable application.

In consequence, software maintenance is an important problem, which can be partially tackled with an appropriate structure of an application. In order to keep the application maintainable for longer, it is necessary to perform anti-regressive work. However, it is not clear if anti-regressive work should be limited to high structural levels or if it should also cover the source code. The next section aims to detail this gap by rephrasing it as research questions.

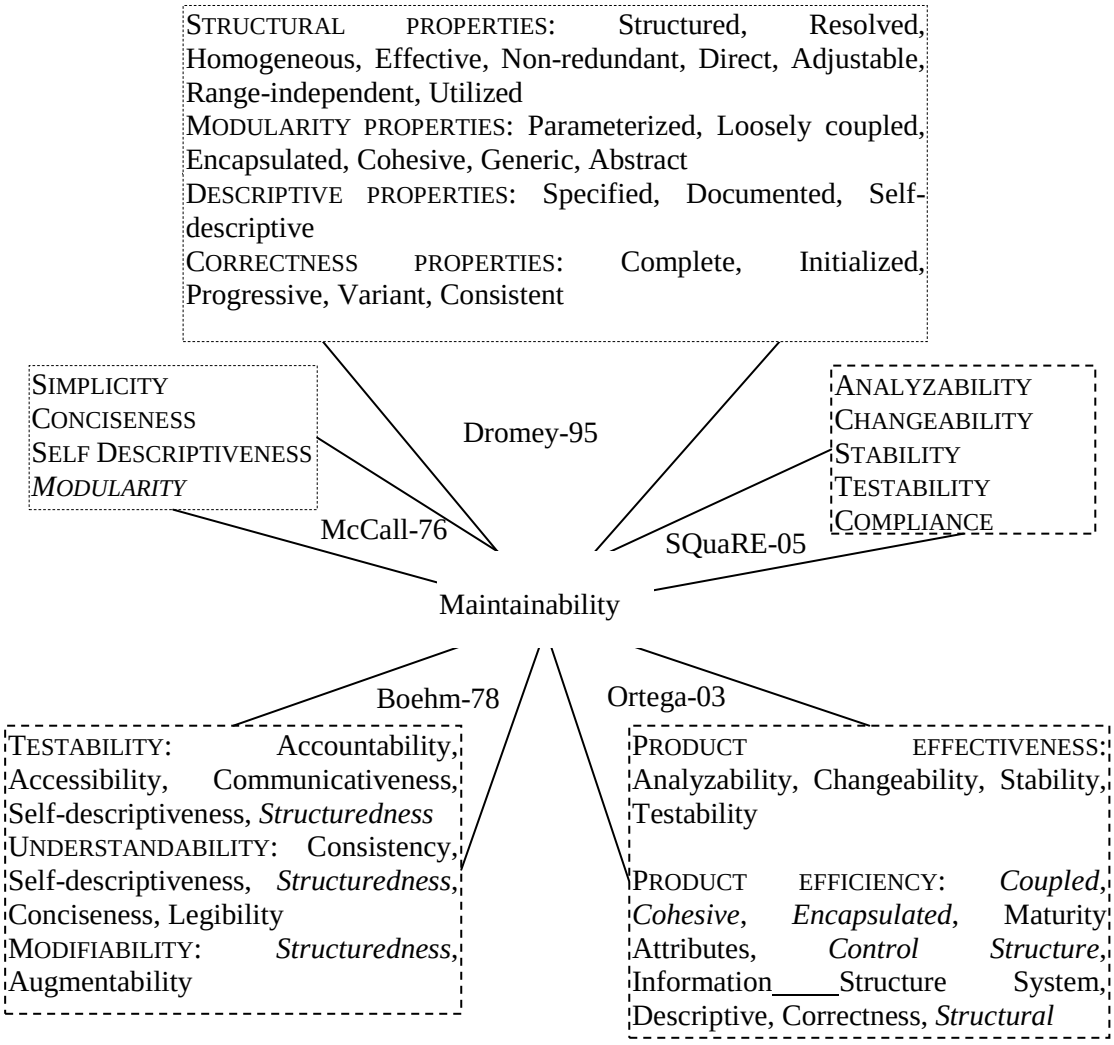


Figure 1-1. Conceptual Models of Software Quality. Factors that affect maintainability. (Factors related to the application's structure are in italics).

1.2 Research questions

There are reasons to believe that an application requires good architectural characteristics such as: modularity, encapsulation, low coupling, high cohesion, reuse, simplicity, etc. to evolve. These characteristics can facilitate the introduction of a change. However, it is not clear yet if it is enough to ensure these characteristics at architectural level, or if it is also necessary to ensure similar characteristics at source code level to improve maintainability. So, the concern that this

thesis tackles is:

Do source code characteristics affect the maintenance of an application?

A metaphor can illustrate this question. In civil engineering, not only the design of a building must be sound but also the materials with which it is built: using low quality concrete can also compromise the lifetime of the building. However, in software engineering it is not clear if the quality of the source code can compromise the evolution of an application, and if so, *which source code characteristics have the worse impact, and therefore, should be an anti-regressive work priority*.

Assuming that software structure impacts software maintenance, minor issues on the implementation may also have consequences on the application's sustainability and longevity. However optimizing the time used for anti-regressive work is a challenging task because it is not clear which source characteristics are harmful and to what extent.

We will use the term **Source Code Issues (SCI)** to refer to source code characteristics that have been pointed out as unhealthy implementation practices. Source Code Issues include: clones (replicated code), god methods (too many responsibilities in a method), god classes (classes that represent more than one abstraction), feature envy (methods that use more fields of other classes than the filed of its own class), etc. Source Code Issues occur at different levels of abstraction. Different types of Source Code Entities (SCE) enclose different SCIs, for instance, god classes occur only in classes, while feature envy occurs only in methods. There are already several approaches to detect SCI, however very little is known about the effect of SCI on the maintainability of the application.

In order to analyze these questions, we investigated the effect of clones on the changeability of methods. Therefore, the research question was refined to:

Is being cloned harmful for the changeability of methods?

This investigation has led us to propose a methodology to analyze the effect of Source Code Issues on the changeability of the Source Code Entities that hosts them.

1.3 Proposed approach

This section motivates and presents a methodology based on empirical observations to analyze the effect of source code issues in maintainability.

1.3.1 Need for a methodology to validate quality frameworks

Quality frameworks are theoretic constructions that aim to assess properties of software applications. One of the disadvantages of quality frameworks is that the quality of the application has a different meaning depending on the stakeholder [Kitchenham '96]. For instance; in terms or software evolution, quality is related to the ease of the application to accommodate major changes; but in terms of the software product, quality is related to the number of faults.

Most quality frameworks attempt to assess the quality of an application based on its internal characteristics. However, this approach has one important disadvantage: several characteristics contradict each other, for instance, high cohesion and low coupling. Therefore, it is not possible to optimize an application's structure to achieve an optimal value in all its internal characteristics. However, it is possible to balance the values among conflicting characteristics. Nevertheless, quality frameworks do not give much insight about how to achieve such balance. In consequence, achieving a high quality application becomes a problem of balancing characteristics without knowing their weight on the quality goal.

Another issue about quality frameworks is that they are theoretical constructions without empirical validation to support their claims. As a result, there is a plethora of quality frameworks proposing similar characteristics with no standard vocabulary, proposing different ways to achieve the same quality goal (see for instance the different decompositions of maintainability above on Figure 1-1). Besides, the lack of evaluation makes quality frameworks incomplete, and probably, inconsistent with real life projects [Moody '05].

Summarizing, quality frameworks explain how to achieve quality attributes but only up to design level. They do not state how to achieve a quality attribute at implementation level [Marinescu '02]. There is no agreement on how implementation heuristics would achieve a quality attribute at design level. Although there are recent quality-models that propose implementation heuristics to achieve a quality attribute [Marinescu '02][Moha '08], different quality-models give different (and in some cases contradictory) heuristics to achieve the same quality attribute [Marinescu '02][Moha '08]. These divergences in quality-models that integrate source code heuristics could be solved by analyzing the effect of not using the heuristics on certain quality attributes of the application.

1.3.2 Proposed methodology

Practitioners are not aware of quality frameworks [Moody '05]. In fact, there is not even a de-facto standard quality framework [Moody '05]. Although most practitioners maintain applications at source code level [Singer '98], it is not known to what extent a source code characteristic that is believed to be harmful affects the maintainability of an application.

We think that it is possible to find facts about the nature of programs from the common results from diverse applications. Therefore, we propose to perform a set of analyses, on several applications, to assess the impact of the SCI. Finding similar results regardless of the applications analyzed, is more likely to be due to the analyzed characteristic than due to random similarities among the applications. Moreover, the greater the number of applications analyzed the smaller the effects of particular characteristics of each application are.

In summary, quality frameworks lack of validation that could be given by empirical methods. We propose an empirically based methodology to assess the effect of source code characteristics that are considered harmful in the maintenance of the application. Our methodology is aimed at researchers in software engineering that want to analyze the effect of a source code characteristic on the changeability of the final application. In particular, when there are reasons to believe that the source code characteristic is harmful. However, it is not necessary to apply the whole methodology for each source code characteristic to analyze. Given that each step tackles different research questions, some steps can be avoided because their questions may have been answered by previous work.

In order to evaluate the methodology proposed, we analyze the effect of cloned code (SCI) in methods (SCE). Clones are duplicated code, and they are believed to be harmful because they may require duplicated maintenance.

1.3.3 Contributions

This thesis presents a methodology to assess the effect of a Source Code Issue on the ease of changing the Source Code Entity that has it. However, the contribution of thesis is not limited to the methodology, but also to the findings obtained by applying it to clones at the level of methods. We confirm that several of the hypotheses about the harmfulness of clones. Firstly, cloned methods change more than code that is not cloned, and that these extra-changes occur inside of the cloned fragments. Secondly, that cloned methods have a higher changeability than

methods without clones. Thirdly, that cloned methods with highest changeability decay are those that are sometimes cloned, possibly due to inconsistent changes.

We have also shown that the nature of the method is critical for identifying methods with high changeability. This is also true for cloned methods, as method characteristics were found as better discriminator of the changeability of cloned methods than clone characteristics.

Finally, we have found that contrary to previous findings, cloning is persistent. Methods cloned, tend to remain cloned all their lifetime.

1.4 Structure of this thesis

The rest of the document is organized as follows. Chapter 2 stresses on the importance of performing empirical analyses to identify which SCI should have priority when performing anti-regressive work. Chapter 3 presents the state of the art in the analysis of clones: their definitions, algorithms to detect them, claimed causes and consequences, empirical results on the causes and consequences of cloning, and open questions in the area.

Chapter 4 explains the steps of the methodology. The methodology is divided into context phases, and analysis phases. Context phases are the ones that create the background needed to be able to answer the research questions, but analysis phases are those that explain how to answer the research questions. Chapters 5 to 7 explain the context phases, and show the results of applying those phases to the analysis of clones. Chapters 8 to 10 explain the analysis phases, and show the results obtained analyzing clones at the level of methods in five open source applications. Chapter 11 presents the conclusions of this work from two perspectives: from the utility of the methodology, and from the usefulness of findings about clones. Chapter 11 also discusses ways in which this work can be extended. A figure explaining the contents of this thesis is shown below (Figure 1-2).

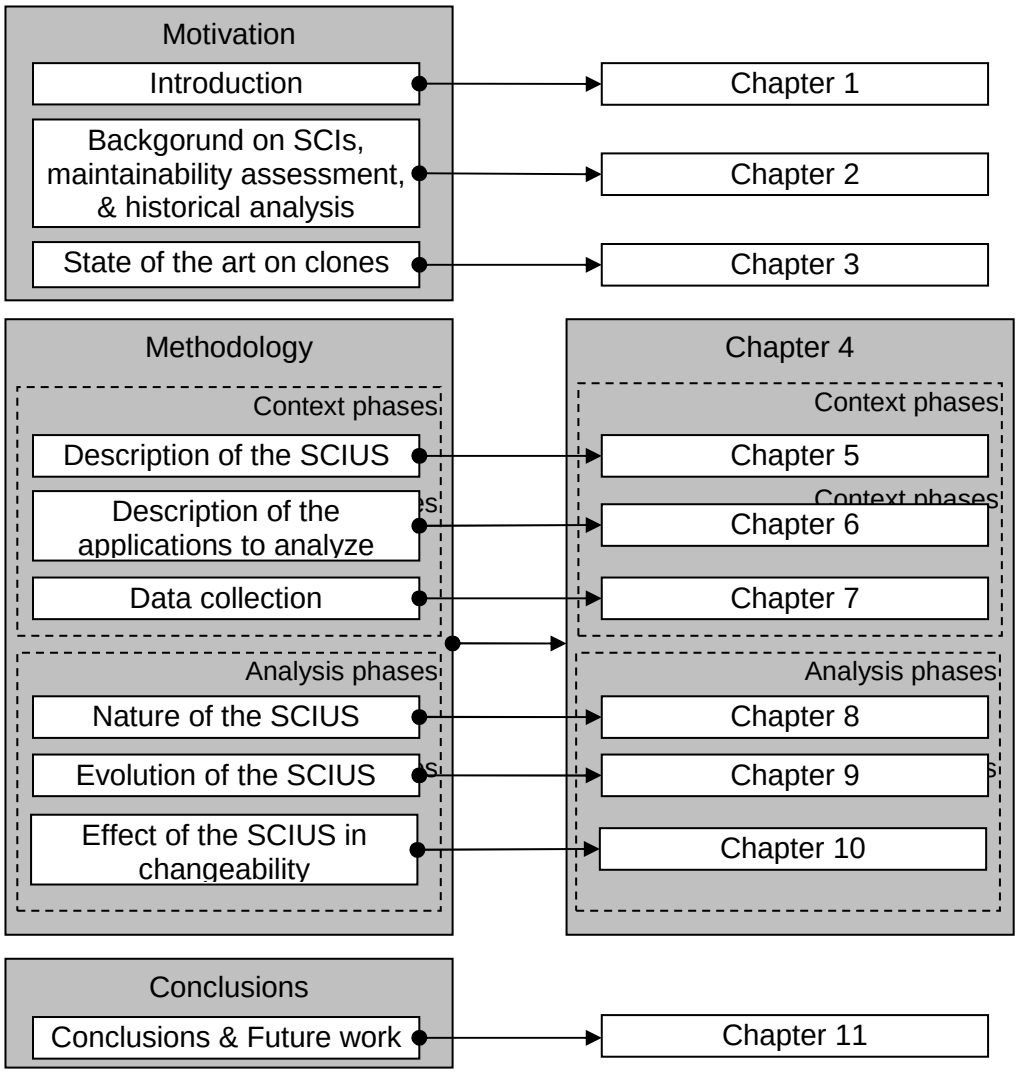


Figure 1-2. Thesis structure

Chapter 2. Background for the methodology

It is important for practitioners to be able to recognize and prioritize source code characteristics that may reduce the ease to change the application. This chapter extends the definition of Source Code Issues (SCI), and argues that the impact of source code issues in source code quality is unknown. The chapter also presents models that integrate source code issues to assess source code quality, and explains how such models would benefit from analyzing the historical impact of source code issues. Finally, the chapter explains why changeability is a suitable proxy to assess maintainability effort, and justifies the analysis of software history to obtain changeability indicators.

This chapter is composed of two sections. The first section motivates this work. The second section justifies methodological choices.

2.1 Source code issues

Source Code Issues are source code characteristics that have been pointed out as unhealthy by the Object Oriented Programming (OOP) community. These source code characteristics have received several names like source code flaws, bad smells, anti-patterns, code smells, etc. We think that the methodology proposed can be used for all of them. Therefore, we decided to propose an umbrella term for these source code characteristics. In order to keep the negative connotation with which they were originally named, we decided to call them Source Code Issues (SCI). Note that a more neutral term like source code characteristics could be confusing.

Common causes of source code issues are misuse of inheritance, missing inheritance, misplaced operations, violation of encapsulation, class abuse [Bär '99]. There are valid reasons to claim the harmfulness of SCIs. However, there is little empirical evidence to support such claims because most research in this area has focused on the definition and detection of source code issues.

This section summarizes previous work on source code issues. It provides a summary of the types of source code issues, the claimed effects of source code issues in maintainability, and the

importance of prioritizing the handling of source code issues that increase maintainability effort in the long term. This section also presents and compares previous analyses on the impact of source code issues.

2.1.1 Classification of source code issues

Given the large variety of source code characteristics that can be considered harmful, we proposed a classification of source code issues (SCIs), depicted in Figure 2-1. There are two types of SCIs the primitives and the complex. Primitive source code issues are those that can be identified at a low level of granularity but their harmfulness cannot be directly linked to higher abstractions in the structure of the application. Complex source-code issues are those that can be directly linked with a problem at a higher level of abstraction than source code. Complex SCIs are usually defined in terms of primitive SCIs, because they accumulate evidence from several source code characteristics and link it with a design problem. In theory, complex SCIs are easier to correct because, by being explicitly related to design, they say why they might be harmful. In contrast, primitive SCIs might be conscientious design decisions, and therefore, they are identified just as refactoring opportunities.

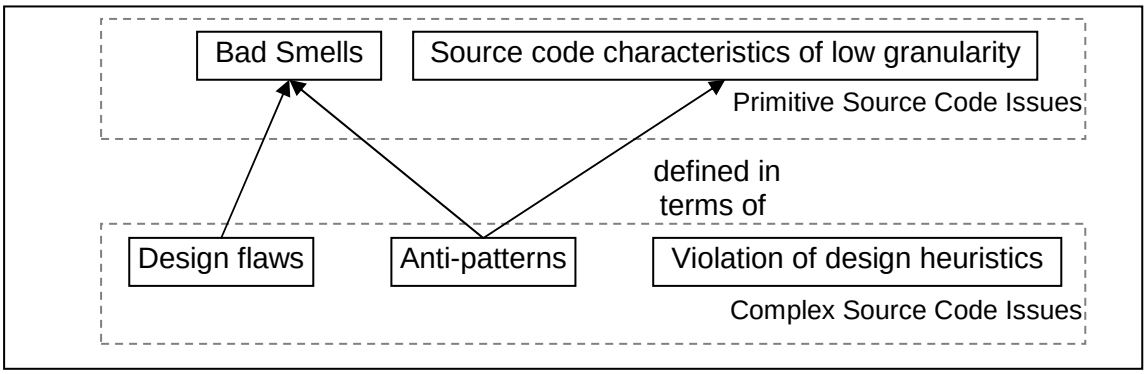


Figure 2-1. Classification proposed for source code issues.

There are two types of primitive SCIs: bad smells, and other low granularity SCIs. Bad smells are a set of characteristics in source code entities that experienced programmers found to be a sub-optimal implementation.

Table 2-1. Bad smells.

Name	Definition. [References to detection strategies]
Comments	When comments do not explain the purpose of the code but the code itself.
Long method	When methods are over sized, they may have multiple responsibilities. [Kataoka '01; Marinescu '01]
Long parameter list	When methods have many parameters, instead of having a few objects that encapsulate the data needed. [Kataoka '01; Marinescu '01].
Cloned code	When the same code fragment is in more than one place. [Baker '95; Mayrand '96; Baxter '98; Mens '03]
Message chains	When a method navigates through the structure of the application.
Switch statements	When switch statements operate on objects of the same type that call similar methods depending on attributes, i.e. they indicate lack of inheritance and polymorphism. ([Emden '02])
Feature envy	When a method uses more methods and fields of another class than of its own class. [Simon '01; Munro '05]
Speculative generality	When a method or a class contains several special cases and unused code that attempts to handle future needs. [Simon '01; Mens '03]
Data clumps	When there are sets of data frequently found together as class members or in method signatures. [Simon '01]
Temporary field	When a class uses its fields only in certain occasions, this may happen because a single class is used to represent several abstractions. [Simon '01; Munro '05]
Primitive obsession	When sets of primitive types are used instead of small objects for dealing with small tasks. This may result in duplicated code, or in scattered concerns.
Large class	When a class has too much code, it might be that the class is doing too much or that there is duplication. [Marinescu '01; Simon '01; Tahvildari '03; Crespo '05; Munro '05; Walter '05]
Data class	When a class does not contain functionality, only fields, getters and setters. This could mean that the set of attributes does not describe an abstraction or that the functionality of the class is somewhere else.[Marinescu '01]
Lazy class	When a class does not have enough functionality to justify the maintenance of a separate abstraction. [Simon '01; Crespo '05; Munro '05]
Middle man	When a class delegates most of its functionality to another class.
Refused bequest	When a subclass does not overwrite the methods inherited from its super-class. In this case it might be better if the subclass and the super-class become siblings that inherit from a new class that represents its commonalities.[Mens '03]
Divergent change	When one a class is frequently changed in different ways for different reasons. Classes with divergent change might be doing many things. [Moha '06a]
Shotgun surgery	When changes affect many classes. [Girba '04b; Xing '04; Moha '06a]
Parallel inheritance hierarchies	Whenever a change to one hierarchy of classes requires the same change to another hierarchy of classes. [Girba '04b; Xing '04]
Alternative classes with diff. interfaces	When methods of different classes do the same thing but have different signatures.
Incomplete library class	When a library misses some methods required or has lots of extra behavior.

There are three types of complex SCIs: violations of design heuristics, design flaws, and anti-patterns. The **violations of design heuristics** [Ciupke '99] occur when the implementation

does not comply with implementation heuristics made to conform to design principles [Johnson '88; Meyer '92; Lakos '96; Riel '96; Martin '00]. Not complying with these heuristics indicates source code that may cause problems to the design. **Design flaws** [Marinescu '01] are source code characteristics that may affect maintainability factors. Design flaws include the composition of several bad smells, the violation of some Object Oriented principles and the lack of use of certain design patterns (like bridge, strategy, singleton, façade, etc.). Another term used for complex source-code issues is anti-patterns. An **anti-pattern** [Coad '91] is a solution for a problem known for being inappropriate. An example of anti-pattern is called spaghetti code, and it is meant for source code that is complex and tangled. Spaghetti code does not use control flow constructs (if, for, while) which makes it more difficult to understand.

2.1.1.1 Bad smells

A bad smell is a set of source code characteristics that indicate an area susceptible of improvement or that could benefit from refactoring[2] [Fowler '99]. For, each bad smell there is at least one sequence of refactorings to eliminate it. According to those who coined the term, bad smells are just useful indicators, but by no means rules, because finding a bad smell does not necessarily indicate that there is an issue; therefore the restructuring decision should depend on the developer. Bad smells may cause the following effects: increase the need to be changed, make the code more difficult to understand, provoke bugs, require similar changes in different parts, and excessive / eliminable code. Note that some effects promote other effects, for instance excessive code makes code more difficult to understand. Below, in There are two types of primitive SCIs: bad smells, and other low granularity SCIs. Bad smells are a set of characteristics in source code entities that experienced programmers found to be a sub-optimal implementation.

Table 2-1, there is the list of the bad smells proposed by Fowler.

[2] A refactoring is a source code-to-source code transformation. Given that refactorings do not alter the algorithm of the program, just its structure, they are usually automatic.

2.1.1.2 Other SCIs of low granularity

There are authors that propose primitive SCIs other than bad smells, such as: low cohesion, multiple interfaces, procedural names, no polymorphism, no inheritance, no parameters, use of global variables, controller methods, and controller classes [Moha '08].

2.1.1.3 Violation of design heuristics

Design principles are key concepts to achieve a good design. A good design is modular, which means that it is made of autonomous elements connected by a coherent, simple structure [Meyer '92]. Achieving a good design is key for maintenance because it 'balances trade-offs to minimize the total cost of the system over its entire lifetime' [Coad '91]. In other words, design principles indicate the structural properties required to consider a design good. Design principles include generic principles and principles that depend on the programming paradigm. Examples of generic principles include generic principles like modularity [Meyer '92], information hiding [Parnas '72; Parnas '79], and separation of concerns [Dijkstra '82]. Examples of principles in Object Oriented Programming include the Liskov substitution principle [Liskov '87], the law of Demeter [Lieberherr '88], and Robert Martin's design principles: open extension-closed modification, dependency inversion, interface segregation, reuse/release equivalency, common reuse, common closure, acyclic dependencies, stable dependencies, and stable abstractions [Martin '96a; Martin '96b; Martin '96c; Martin '00].

Design heuristics are implementation procedures to increase the probability of complying with design principles. Given that design heuristics are defined at source code level, their violation may indicate the disobedience of design principles. Therefore, violating design heuristics may pose a threat to maintainability.

2.1.1.4 Design flaws

Three models relate source code characteristics to maintainability issues. Marinescu's model that merges the information of design flaws and source code maintainability through metrics [Marinescu '04b]. One side of the model decomposes each design flaw into primitive SCIs, and each SCI into a set of structural metrics. The other side of the model decomposes maintainability into maintainability factors, and maintainability factors into metrics. Tracing which SCI affects which maintainability factors is done through the structural metrics, which are the common factor in both sides of the model.

Tahlvidari and Kontogianis [Tahvildari '02] propose a maintainability soft goal-graph that indicates how certain high level source code transformations would improve or worsen maintainability factors of an application. The soft goal-graph would show which transformation maximizes the maintainability, where the maintainability is assessed using structural metrics i.e. a reduction on the values of the metrics indicates the presence of a source code issue.

Table 2-2. Design flaws.

Name	Definition [References to detection strategies]
God Method	God methods centralize the functionality of a class, and it is characterized by an excessive size and complexity [Marinescu '02]
God Class	God-Classes centralize the functionality of an entire subsystem. A god class uses the data from other classes, and delegates only minor details to a set of trivial classes. [Marinescu '02]
God Package	A god-package is a package large and non-cohesive that has a large number of clients that use it excessively. [Marinescu '02]
Misplaced Class	A misplaced class occurs when a class needs more classes of other packages than classes of its own package. [Marinescu '02]
Wide Subsystem Interface	It occurs when the interface of a package is very wide so there package and its clients become tightly coupled. [Marinescu '02]
Lack of Bridge	It occurs when inheritance is used instead of using a bridge pattern, which derives in an inflexible solution because the abstract super-class and the concrete sub-classes are statically linked. [Marinescu '02]
Lack of Strategy	It occurs when solving a problem may require a family of interchangeable algorithms. However, instead of using the strategy pattern, the algorithms are implemented in a single class or in a hierarchy of classes where the sub-classes override the implementation of the super-class. [Marinescu '02]
Centralized control	It is a violation of even distribution of data and behavior among the classes. It is necessary to find feature envy to consider that there is centralized control. [Trifu '05]
Inheritance for usage	It is a violation of using inheritance only for specializing the behavior of the super-class. It is necessary to find refused bequest to consider that there is inheritance for usage. [Trifu '05]
Schizophrenic class	It is a violation of one class represents only one abstraction. It is necessary for the class to be large and complex (god class) to consider it schizophrenic. [Trifu '05]
Explicit state checks	It is a violation of one class represents only one abstraction. It is necessary to find god classes with temporary fields to consider it explicit state checks. [Trifu '05]

Design flaws, defined in [Marinescu '04b], are used [Trifu '05] in a metaphor to reason about source code quality: design flaws are symptoms of source code illnesses, and a source code illness is a violation of Object Oriented Design Principles. For instance, abusive conceptualization is an illness that occurs when the ‘one class = one abstraction’ principle is broken. There are three types of illnesses related with three areas of the object-oriented paradigm: definition of concepts, distribution of intelligence among the concepts, and

distribution of intelligence among inheritance hierarchies. In order to diagnose an illness, it is necessary for some symptoms to be present, while other symptoms may not occur even if the source code entity suffers the illness, i.e. some design defects are mandatory to consider that a source code entity has violated a design guideline. The three types of illnesses (violations of basic design principles) are defined by ten types of symptoms (design flaws). By making explicit the illnesses that generate violations of design guidelines, the author creates a higher abstraction of structural problems, which also points out possible ways of 'treating' the illness.

2.1.1.5 Anti-patterns

Moha and Guéhéneuc propose a quality framework oriented to the categorization and identification of problems as anti-patterns which also offer a higher level of abstraction to reason about structural issues in source code [Moha '06b]. Each anti-pattern is decomposed at several layers of abstraction into primitive source code issues, and each source code issue is divided in the properties that characterize it. For instance, spaghetti code is characterized by the following source code issues: procedural names, long methods, methods without parameters, global variables, no inheritance and no polymorphism.

Table 2-3. Anti-patterns as defined in [Brown '98].

Name	Definition [References to detection strategies]
Blob	It occurs when too many functions are concentrated in a single part of the design, usually a class. A blob is a large class that centralizes most of the processing done by a program. The blob declares many fields and methods with a low cohesion among one another. [Moha '08]
Spaghetti code	It occurs when there is lack of structure in a system due to misuse of language abstractions. In the case of OO code, it occurs in applications that do not use inheritance, polymorphism, or other reuse abstractions offered by the paradigm. Spaghetti code is characterized, in OO applications, by few classes with large methods that are called once in the application, low interaction among objects, and methods without parameters that rely on global variables. [Moha '08]
Functional decomposition	It occurs when methods are converted into classes, so the functionality is implemented by a main method that calls many helper methods. It is characterized by complex code, inexistent class hierarchy, and classes that offer just one functionality. [Moha '08]
Swiss army knife	It occurs when a class offers a high number of services to address many different needs. Although may functionalities are offered, only few of them are used in the application. [Moha '08]
Poltergeist	Poltergeists are objects whose purpose is to pass information to another object. They are characterized for having a short live, having no state, and being used to perform initializations or to invoke methods other classes. [Moha '08]

2.1.2 Assessing the impact of source code issues

This section shows the lack of work in this area, and discusses why the evidence so far is not useful yet to support the software development process.

The impact of source code issues has been analyzed in four ways:

- the extension of the source code issue (what percentage of the application is affected)

- the persistence of the source code issue (how long is the lifetime of the source code issue)

- their relationship with changes

- and their relationship with bugs.

Analyzing the extension of the source code issue in the application can be done in several ways. By reporting the size of the application (in terms of modules, files or functions) and the number of modules, files or functions affected by the source code issue [Lague '97]. By checking which percentage of the application is affected by the source code issue [Ducasse '99]. By checking if the percentage of the application affected by the source code issue is localized in certain modules, files or functions [Chou '01; Antoniol '02].

The persistence of the source code issue in the application refers to the average lifetime of the source code issue in the application. It is analyzed because several researchers think that volatile source code issues, i.e. those that are eliminated soon after they are inserted in the application, are harmless because they did not have time to make any difference in the application. There are several ways to measure persistence. Chou et al. measured the average number of days and years that the SCI is to the application [Chou '01]. Ratiu et al. measured the percentage of the source code entity's lifetime during which the source code issue is located [Ratiu '04]. Antioniol et al. and Lague et al. measured the percentage of application affected by the source code issue over time [Lague '97; Antoniol '02]. Chou et al. also checked if the age of the source code is related with the amount of source code issues that it has [Chou '01].

Source code issues are believed to increase the number of changes and the number of bugs. Therefore, several of the studies about the impact of SCIs aim to find correlations between them and number of changes, or number of bugs. However, it is important to remember that lack of changes does not necessarily means that the source code issue is harmless: it could happen that the code with the source code issue is so complex that no developer dares to touch that code.

This phenomenon is called code viscosity [Martin '00].

Regarding the relation between source code issues and changes there are two kinds of experiments: those that aim to show that SCEs change more if they have the SCI, and those that aim to show that SCEs having related SCIs tend to co-change i.e. change at the same time. To show that SCEs with a SCI change more than other SCEs, authors have used as proxy of changes the number of releases of the file [Monden '02], and the number and frequency of changes that the entity has undergone [Ratiu '04]. Showing that the SCI increases the co-changes is done for SCIs that relate SCEs, for example, clones or feature envy. The relation between SCIs and co-changes has been studied by trying to correlate files that share the same SCI with files that co-change [Geiger '06], and trying to measure if the SCEs that have related SCIs are changed in the same way at the same time [Lague '97; Kim '05; Krinke '08] (or at a close time interval [Aversano '07]).

Finally, regarding the relation between source code issues and bugs, researchers have tried to correlate the size of the source code issue with the number of bugs [Fenton '00; Monden '02], and to show that some bugs are originated from inappropriate handling of SCIs [Li '06; Aversano '07].

These approaches cannot be used to recommend what to do with a Source Code Issue. For instance, we do not know if, in general, clones make methods more difficult to maintain or not. It is necessary to have an approach that takes into account several definitions of harmfulness, and that weighs these definitions to be able to compare different SCIs, and finally that provides ways to predict which characteristics make a SCI more difficult to maintain.

2.1.3 Automatic elimination of source code issues

Designing code flexible to changes is a difficult task. “Although there is no 'best' structure for a solution to a given problem, there are definitely better and worse structures—the distinction is typically made largely as a matter of intuition combined with experience [Darcy '05]”. A flexible design structure is usually achieved through refinement after several incremental iterations because diverse possible modularizations of a program may be conflictive, but only one can be chosen. That one must support most of the foreseen changes [Griswold '93].

Moore’s tool [Moore '96] was the first approach to eliminate source code issues automatically by extracting shared expressions from methods into a new hierarchy.

Recently, the interest for correcting and improving automatically source code structure has grown again [Keeffe '04; Trifu '04; Seng '05]. Tahvildari and Kontogianis propose to analyze the overall effect of introducing high-level refactorings (using patterns) in maintainability factors. The refactoring chosen depends on the cost of introducing it and the benefit on maintainability factors [Tahvildari '02; Tahvildari '04]. Keeffe and Cinneide [Keeffe '04] propose a set of metrics to express design heuristics. The improvement is achieved by repeatedly applying a random refactoring to the design, and maximizing the quality of the obtained design, calculated as the weighted sum of metrics. Seng et al. [Seng '05] use genetic algorithms to evaluate the best structural arrangement using metrics they propose themselves to measure the overall compliance of the architecture with design heuristics. Although correction techniques look very promising, it has been shown [Moore '96] that automatic transformations can have a negative impact on software comprehension given that they undermine the mapping between problem domain concepts and software entities.

2.2 Measuring maintainability

This section motivates the use of historical analysis of software to assess changeability as a proxy for maintainability. The first part of the section introduces maintainability terminology, and explains why changeability can be used to assess maintainability. Then the section shows two approaches to measure maintainability and changeability: by analyzing the structure of the application or by analyzing the changes the application has undergone. Finally, this section argues that historical analysis of software is an appropriate method to assess changeability and maintainability.

2.2.1 Maintainability terminology

Maintenance is any activity performed on a system after the first delivery of the application [Bennett '00]. Maintenance can be divided into several stages: evolution, servicing, phase-out, and close-down [Bennett '00]. Evolution is the stage in software lifecycle in which the system is subject to fault correction and to adaptation to new user requirements or to a new environment. Evolution is a term also used for describing the dynamics of software history [Lehman '85]. Servicing is when only minor changes are possible, phase-out when no changes are performed [Bennett '00] and close-down is when the system is not used anymore [Bennett '00]. These phases of maintenance are depicted in Figure 2-2.

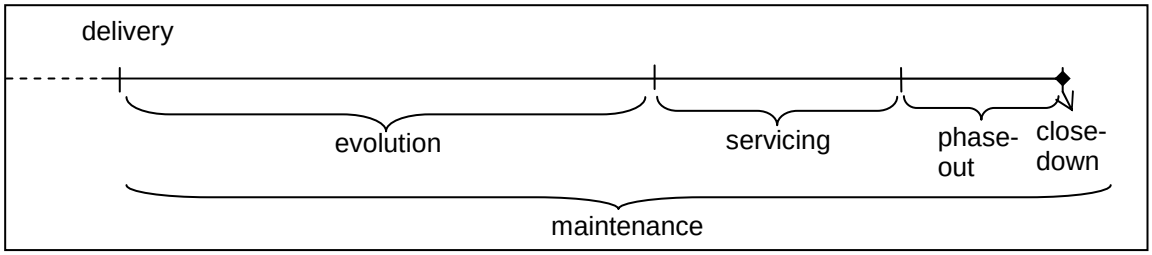


Figure 2-2 Phases of maintenance.

Traditionally, software maintenance tasks have been classified in four categories depending on their goal: adaptive, corrective, perfective, and preventive (see Table 2-4). **Adaptive maintenance** keeps a computer program usable in a changed or changing environment [Lientz '81; IEEE '99]. **Corrective maintenance** corrects discovered faults [Lientz '81; IEEE '99]. **Perfective maintenance** improves the performance, maintainability, or other attributes of a computer program [Lientz '81; IEEE '99]. **Preventive maintenance** detects and corrects latent faults before they become faults [IEEE '99].

Table 2-4. Classification of types of maintenance tasks according to (ISO/IEC 14764)

	Correction	Enhancement
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

While maintenance is a phase of an application's lifecycle, maintainability is a property of the application that permits a longer evolution period (i.e. it can accommodate significant changes retarding the servicing phase). **Maintainability** is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications [ISO '01].

2.2.2 Changeability as a proxy to assess maintainability

Maintainability has not only been regarded as a property of the application, but also as the ease to perform a phase of the application's lifecycle (maintenance). Definitions of maintainability like the "effort required to locate and fix an error in an operational program" [Gaffney '81], or "the ease with which an application or component can be maintained between major releases" [Firesmith '03], show that maintainability has been also used in the sense of effort measurement.

According to the framework for software quality 'Software Product Quality Requirements and Evaluation' (SQuaRE - ISO 25000) [ISO '01], maintainability is achieved through the following factors: analyzability, changeability, stability, testability, and maintainability compliance. **Analysability** is the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified [ISO '01]. **Changeability** is the capability of the software product to enable a specified modification to be implemented [ISO '01]. **Stability** is the capability of the software product to avoid unexpected effects from modifications of the software [ISO '01]. **Testability** is the capability of the software product to enable modified software to be validated [ISO '01]. **Maintainability compliance** is the capability of the software product to adhere to standards or conventions related to maintainability [ISO '01].

Changeability has also been defined as the "attributes of software that bear on the effort needed for modification, fault removal or for environmental change" [Sanders '95]. Notice that changeability and maintainability have very similar definitions. Changeability would be a pertinent proxy of maintainability because changeability is the characteristic that allows performing changes with a low effort. However, changeability does not represent all the characteristics that an application should have to be maintainable. That means that changeability is necessary but not sufficient to assess maintainability. Therefore, if the presence of SCIs is correlated with changeability, one can conclude that the SCI affects maintainability. However, if there is no correlation between the SCI and changeability, it does not mean that the SCI does not affect maintainability.

2.2.3 Structural assessment of maintainability

Most of the early work on maintainability assessment is based on the application's structure. Source code maintainability is divided into factors that may impact it, such factors are evaluated using questionnaires and measures by all those involved in the development, and finally the results of the questionnaires are merged into statistical models that weigh the impact of each factor [Peercy '81]. Although the method uses measures, they are not recommended as absolute command but as guides to identify possible causes of maintainability problems [Peercy '81]. There are two problems with this approach: first the evaluations are subjective, and second there is a high variability between evaluators [Bennett '93].

Thanks to the observation that maintenance on code that is difficult to understand is non-trivial, and that the difficulty in understanding it could be a consequence of the code's complexity, some approaches focused on calculating the complexity of the code. Complexity metrics like Halstead's [Halstead '77] and McCabe's complexity measures [McCabe '76]) were proposed to quantify the difficulty experienced by the maintainer. Later, the assessment of maintainability was extended to aspects other than program comprehension, so metrics based on static analysis of the source code to predict maintenance costs were proposed [Oman '92; Welker '95]. From these metrics, the **maintainability index** is the one more widely accepted because it can be measured along the process with simple tools. The maintainability index is defined as a polynomial formula of the number of operands and operators, the number of branches, the number of lines of code, and optionally the percentage of comments, all of them at module level [Welker '95]. The maintainability index is considered useful to track code quality, to find areas that could reduce lifecycle costs, to find modules of high risk to modify, and to compare in-house applications vs. third party's applications [Coleman '94]. However, this approach has two problems: first it is inconclusive on predicting the maintainability of a system, and second it cannot be used as universal measurement because it can only signal trends of the software to be better or worse in terms of maintainability [Bennett '93].

Nevertheless, not all structural approaches to assess the maintainability of code are based on traditional source code metrics. Assuming that a complex call structure implies a more difficult maintenance, Burd and Munro [Burd '99] propose a measurement based on the complexity of call relations, defined as the number of structural paths for getting from one function to another. If the number of functions that can be called just by one path increases, then the maintenance is considered simpler (inversely when the number of nodes decreases). They measured how the complexity of the call-structure changed from release to release, and validated their predictions on easier/worse maintenance by interviewing maintainers. In addition, the complexity of call relations of a release was compared with the proportion of types of change (adaptive, perfective, corrective or preventative) in that release. The comparison indicated that the more preventative maintenance there is in the release, the more functions were called only by one path. The authors concluded that the measurement proposed seems to reflect some characteristics of code maintainability. Bianchi et al. [Bianchi '01] also propose a measurement based on the degree of disorder among its components, that they call entropy. Entropy measures maintainability because it assesses the amount of time required to track all the components involved in a

change. The authors found that entropy increases over time among all the sub products of software.

2.2.4 Historical assessment of maintainability

Another approach for measuring maintainability is by quantifying the effort required for implementing the changes. In the strict sense, it is not a maintainability assessment given that historical analysis cannot always measure other factors that affect maintainability such as analyzability or testability. However, as mentioned before, maintainability has been regarded also as the effort required to implement changes. A more accurate label for this approach would be 'changeability assessment'.

Assessing the effort of performing a change from the history of an application is difficult because not only the implementation of the change should be taken into account but also all the activities related to the change. Besides, related activities of different changes may overlap [Graves '98]. For instance, a developer may be identifying which places require modification for one change, while ensuring that another change does not affect functionality that was working. Graves and Mockus [Graves '98] measure changeability based on the assumption that worse code requires more effort to be modified. They used several predictors of effort and then tried to fit the predictors to a curve. They found that the best predictors were the size of the change, the type of change, and the date where the change was done, but neither the developer nor the time interval that took to implement the change were found as accurate predictors. They found that each year the difficulty of implementing changes increases by 20%, and that those changes that fix faults require 80% more effort than those that do additions of new functionality.

Eick et al. [Eick '01] attempted to measure whether code became more difficult to change over time. They defined **changeability decay** if the time needed for changing the code increases, or the quality of the code decreases. They tested several predictors of changeability decay such as the number of files changed, the size of the files modified, the number of lines added and deleted, the time interval that took to perform the change, the number of developers involved on implementing the change, and the number of lines of code changed. In this case, neither the sizes of the files changed nor the number of developers predicted changeability decay. Nevertheless, they also found evidence of changeability decay over time: the modularity decreases, the span of changes increases, new changes introduce bugs, and time and change

span are correlated with changeability decay. They conclude that changeability decay is a consequence of changes rather than a consequence of complexity.

Van Belle [Belle '04] defines the effort of modifying a source code element as the impact of modifying such element multiplied by its likelihood of changing it. The likelihood is the probability of a source code element being involved in a change. The impact is the expected change size, if a source code element changes. Here a change is the set of modifications in one day, and the change size is the number of source code elements changed. The impact and likelihood of change of diverse code entities (interfaces, classes, etc.) were measured to quantify the usefulness of encapsulation. The results [Belle '04] show that change on abstract code entities (such as interfaces) is infrequent but affects many other entities; while change on concrete code entities (like method bodies) is frequent and has a very low impact on other code entities.

Arisholm and Sjoberg [Arisholm '00] compare structural metrics vs. metrics of the complexity of the change to assess changeability decay. They found that metrics of the complexity of the change capture better changeability decay, but that neither structural metrics nor change complexity metrics reflected several aspects of changeability decay. Hassan and Holt [Hassan '04] also found that historical co-change is better predictor of change propagation than structural dependencies. Finally, Girba et al. [Girba '04a] found that historical co-change, weighted by how recent changes between two source code entities are, can be used to predict future changes. Nevertheless, this approach does not take into account the fact that source code entities are renamed and moved over time.

Summing up, there are two ways to measure maintainability: using structural properties of the application, and characterizing previous change on source code entities. The first approach assumes that the complexity of the source code entity influences the difficulty of maintaining that source code entity. The second approach assumes that change characteristics describe the difficulty of implementing previous changes and that the effort is related with the source code entity changed. Although the historical approach has more assumptions, it is more direct to assess effort. Moreover, empirical studies that compare structural and historical approaches have found that historical approaches are better predictors of changes and their required effort.

2.3 Analysis of software history

Software history is empirical evidence that can be extracted from configuration management systems. Among the information that can be extracted from configuration management systems there is the history of changes, the items affected, the rationale about change decisions, authoring information, etc. Historical information about software has served to characterize how applications evolve. Examples of this usage of historical information are: software aging [Bianchi '01] [Eick '01], the reconstruction of the evolution process [Demeyer '00; Zou '03], and evolution laws [Lehman '97]. Furthermore, historical information about software has also served to support future changes of the application. For instance, the identification of stable and unstable areas in code [Girba '05], the identification of dependencies across the code [Zimmermann '03; Zou '03; Girba '04b], and the extraction of information to predict future changes [Girba '04a; Hassan '04]. Nevertheless, little attention has been given to a deep understanding of the impact of source code characteristics on evolution.

This section explains in detail, how historical analysis is currently done, the type of findings that historical analysis has achieved, and that metrics proposed for historical analysis can be used as changeability indicators. This section is divided into four parts. The first part explains how software history is traditionally obtained from code versioning repositories. The second part presents the studies that try to understand the dynamics of software evolution. The third part shows how software history can be used to identify source code issues. Finally, the fourth part shows how software history can be used to propose improvements of the source code.

2.3.1 Software history extraction

Although a configuration management system can be used to know which changes were done together, which files, methods, classes were part of the application at a given moment, or which files, methods, classes were changed, why, when, by whom, this information is not explicitly stored. It is necessary to process the information stored in configuration management systems in order to get the history of changes of an application. A **System Configuration Management** (SCM) application stores several versions of the same file. Developers usually work on their local copies of the source code and periodically upload changes to different files to the server that has the SCM. The SCM applications automatically save and merge the deltas between the uploaded version and the version in the server.

Knowing which changes were done together (a **commit transaction**) is useful to identify logical changes [Gall '98], e.g. the source code entities that are related in achieving a requirement. However, some SCM applications do not store the information about commit transactions, making necessary to reconstruct them. Two similar approaches were proposed to recover commit transactions [German '04; Zimmermann '04], both of them focusing on a popular SCM application called CVS. Although CVS does not store which sets of files were uploaded to the repository in the same transaction, it saves all the information separately for each file: author, message, and timestamp of the end of the upload. The changes that were done in the same commit transaction would have the same author and message, and similar timestamps. Changes are ordered from the earliest to the latest in terms of timestamp. Most of the changes that are close to each other would have the same author and message indicating that they are likely to belong to the same commit transaction. However, in some cases, changes would seem intertwined. In order to recognize if the changes that seem intertwined are part of a large commit transaction, or if they are a single small commit transaction, it is necessary to group changes by time intervals. The difference between the two approaches proposed relies on how to establish the threshold among timestamps to consider changes within the same commit transaction. In other words, the issue is to decide when the commit transaction finishes. Zimmerman and Weissgerber [Zimmermann '04] propose time windows: fixed or sliding. A fixed window is a pre-set time interval; it assumes that an upload operation will take, at most, a fixed amount of time. A sliding window does not assume any interval of time, a change belongs to the commit transaction if it happened at most three minutes after the timestamp of the last change that was identified as part of the commit transaction [Zimmermann '04]. German's approach is to define a minimum interval between timestamps (45 secs) and a maximum interval for the whole commit transaction (600 secs) [German '04].

2.3.1.1 Identification of methods

Identifying the methods of the application implies recognizing the structure of source code entities that compose an application until the level of methods, as Figure 2-3 shows. This means that it is necessary to store which class each method belongs to, which file each class belongs to, and to which module each file belongs.

The identification of the structure of SCEs that compose an application can be done either by syntactic analysis [Aho '74] or by lexical analysis [Murphy '96]. Syntactic analysis provides a

very accurate idea of the SCEs and their relations because it builds an instance of the programming language grammar using source code files; this means that it verifies the correctness of the source code. An alternative to syntactic analysis is analyzing intermediate code (like bytecode) instead of source code. Lexical analysis identifies special tokens to locate the SCEs in a file, without checking that the source code file indeed complies with syntax rules. Lexical analysis provides information of the packages that compose the application, the source code files of each package, the name of the classes of each source code file, the signature of the methods of each class, and the name of the variables of each class.

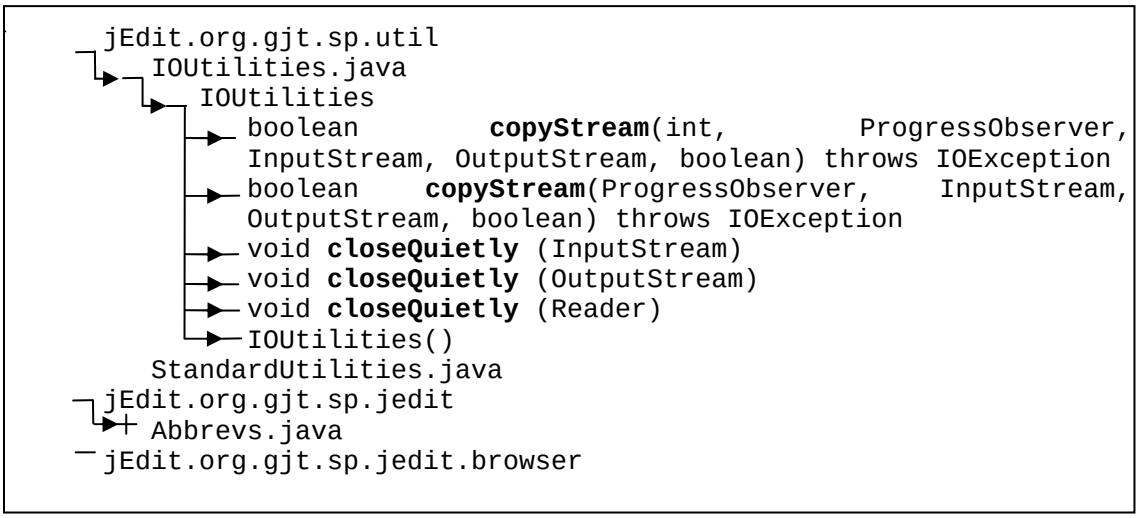


Figure 2-3. Example of the structure of SCEs required for this adaptation of the methodology. The level of the SCE is indicated by its indentation.

There are several considerations for selecting which approach should be used to gather the data for the methodology. First, the tool used should be fast so the results of gathering the data for a large amount of the logical changes can be obtained in a reasonable period. Second, it is desirable that the approach used take as input the data stored inside the SCM repository without additional processing. Third, the approach chosen should be reliable over changes that the applications to analyze may undergo over a large period.

Based on these considerations, there are several reasons for which syntactic analysis is not an advantageous option to identify the structure of SCEs. First, syntactic analysis requires source code files that are syntactically correct, which is not the case for all snapshots stored at SCM repositories. This means that whenever files are syntactically incorrect, the structure of SCEs obtained is incomplete or incorrect. Second, syntactic analysis depends on the grammar of the

programming language, and the grammar of the programming language may evolve. For instance, the grammar of Java has changed twice in the last 10 years (in the period from 1999 to 2009) [Gosling '96; Gosling '00; Gosling '05]. Having a new grammar in the programming language imposes new versions of the syntactic analysis tool, and additional processing to recognize the version of the programming language grammar in which the source code is written. Third, syntactic analysis based on intermediate code would require additional processing to compile the source code entities stored in the SCM repository. This process may not be always successful because SCM repositories do not check the correctness of the source code files stored. Besides, this option is only available if all the applications to be analyzed are written in a compiled programming language, which is a programming language that requires a virtual machine to be interpret and execute the intermediate code generated when the source code is compiled e.g. Java.

Conversely lexical analysis suits well the detection of the structure of SCEs because it only requires the source code files, is faster than syntactic analysis, and its does not depend on the correctness of source code files.

In conclusion, finding which source code entities compose the application is usually done by a lexical analysis of the source files [German '04; Zimmermann '04], because it is the approach that presents the best time performance. Given that historical analysis requires iterating for each commit transaction, operations that take the shortest amount of time are preferred.

However, identifying the commit transactions and the source code entities changed in those commit transactions is not enough to gather an accurate history of the application. The identification of changes in source code entities may be inexact because it is based on the name and location of the source code entity, which may change over time: for example, methods are moved to different files, and they can acquire or loose parameters as they evolve. Being able to identify source code entities over time regardless of changes in their name or location is called **origin analysis**. The next subsection explains the main approaches proposed to achieve origin analysis.

→ *Tracking of methods across snapshots*

Methods may be renamed or moved along their lifetime; therefore, tracking methods across snapshots is an important step of the data collection phase to obtain accurate data for the adaptation of the methodology at the level of methods. Detecting if two methods of different

snapshots are the same method is called origin analysis. In order to detect the origin of a method (i.e. its version on a previous snapshot), one should compare their `identifiers`. To compare the identifiers of two methods, one should define the identifier of the method, and the function to compare identifiers.

Table 2-5. Approaches to perform origin analysis.

Approach	Identifier	Comparison/Similarity function
Mixed [Godfrey '02; Godfrey '05]	Name	Length of longest common substring, over length of the method name plus length of the candidate name
	Declaration	Lexical similarity between: names of the parameters, types of the parameters, and both (names and types of the parameters).
	Set of structural metrics	Euclidean distance of 5 metrics[Godfrey '02] Predefined intervals for each metric. For each interval, the metric has a similarity value (0, 0.1, or 0.2, being 0 the least similar). The values of the metrics are added to give a number between zero and one.
	Sets of callees and callers	Overlap of the relations. Intersection of callers & callees related to the candidate and the method; divided by the no. of callers & callees related to the candidate, plus no. of callers & callees related to the method
Clones [Rysselberghe '03]	Clone detection by lines of code	Visual comparison
Names & Relations [Xing '05]	Name	Pairs of letters shared, over total pairs of letters
	Relations	No. of times that candidate and method have a relation with a third method the same amount of times, over No. of third methods that are related with the method and the candidate
Keywords [Antoniol '04]	Vector of words that discriminate a class from other classes.	The similarity is the cosine of the angle between the corresponding vectors. If the similarity is over a threshold, the class replaced the candidate.
Metrics [Demeyer '00]	Structural metrics	The difference of metrics of the class or method, and the candidate is positive, negative, or zero. Only size and inheritance metrics are taken into account. Depending on the increase or decrease in certain inheritance and size metrics it can identify merge, split, and move scenarios.

Several proposals for fingerprinting methods exist: the name of the method [Xing '05], the lines that compose the method [Rysselberghe '03], set of names of variables that discriminate a method from others [Antoniol '04], etc. Note that some proposals can use the same identifier but they can compare the similarity of those identifiers differently. Furthermore, some origin analysis algorithms use several definitions of identifier to increase their accuracy. To clarify the approaches proposed so far, Table 2-5 indicates for each algorithm the identifier and similarity function used.

Once the possibilities for origin analysis have been reviewed, it is important to weigh the advantages and disadvantages of each approach. It is important to keep in mind that origin analysis must be done for each logical change that has methods that seem eliminated. This means that each code identifier must be obtained for all methods that seem new, and all origin candidates for those methods (i.e. all methods that seem deleted). These constraints imply that the approaches proposed for origin analysis are not ideal for the methodology; their issues are discussed in detail below.

The *mixed* approach [Godfrey '02; Godfrey '05] has high accuracy, but it might be very costly in terms of time because of its need to calculate several identifiers. Besides, this approach requires obtaining identifiers that are costly in terms of processing time and in terms of memory, like the identification by called and caller methods. The *clones* approach [Rysselberghe '03] cannot be used for automatic analysis, and therefore it is useless for the methodology. Furthermore, any approach based on clone detection would require a fine calibration on the level of dissimilarity tolerated. Besides, the algorithm should be able to distinguish the origin from a candidate that has a similar fragment of code (i.e. a clone). The *name and relations* approach [Xing '05] has the same problem of the *mixed* approach [Godfrey '02], that is the cost of calculating for each one of the methods analyzed, the methods it calls and the methods that call it. The main problem of the *keywords* and *metrics* approaches [Demeyer '00; Antoniol '04], is that they are not designed to find the origin of methods. The *keywords* approach [Antoniol '04] requires finding all the words/identifiers in all methods in order to find the words that discriminate one method from the others. The fact of expanding the analysis from the set of methods deleted and created in a logical change to all methods alive before and after the logical change, make of the origin detection by keywords [Antoniol '04] inadequate for the methodology. Finally, the *metrics* approach [Demeyer '00] can identify methods that were moved to other classes but not which methods, or that a method was merged from other methods but cannot identify whenever a method just changed its name, or when some parameters were added.

In summary, the existing approaches take too long to be calculated which make them unsuitable for applications with large histories.

2.3.1.2 Identification of changes

Reconstructing the history of changes of an application does not only include identifying the commit transactions, but also identifying first which source code entities were part of the

application, and then which ones were modified at each commit transaction. Finding which SCEs changed is done by comparing consecutive commit transactions, and finding the changes between them. To compare two commit transactions it is necessary to follow five steps. First, downloading the files changed on the initial commit transaction. Second, finding the SCEs defined in the files that compose the initial commit transaction. Third, downloading the files changed on the consecutive commit transaction. Fourth, finding the SCEs defined in the files for the consecutive commit transaction. Fifth, finding the differences between the source code entities of the initial and the consecutive commit transaction. The comparison can be used to obtain the set of SCE added (those that are just in the latest commit transaction), the set of SCEs removed (those that are only in the earliest commit transaction), and the set of SCEs changed. To identify the set of entities changed, it is necessary to check if the differences between both versions of each file correspond to the lines of any of the source code entities that are in both commit transactions. This approach to find the source code entities changed from one transaction to the next assumes that all the changes are stored in the same history line. However, that is not usually the case. When it is necessary to keep a stable version of the code, the history of the application is divided: the main history line (trunk) may have several branches. Changes to the same files may happen in branches and in the trunk at the same time. Therefore, to be able to identify correctly the changes on a file it is necessary to analyze the commit transactions of each branch separately, or analyzing only the trunk.

2.3.2 Analyzing software history as a phenomenon

This section presents some of the studies that aim to discover the rules that govern the evolution of software applications. The first work in this regard was done by Lehman and Belady [Lehman '85]. They proposed to make measurements of an application along its development process (per release and per day). Among the aspects they measured are: number of developers, number of modules, number of instructions per module, number of modules touched (i.e. changed), number of new instructions per module, etc. Their purpose was to describe the dynamics of the evolution of an application. They found that software continuously changes and that although the application grows in size, its complexity also grows. These observations were interpreted in a theory of software evolution that defined three types of systems, depending on their likelihood to require changes, and proposed a set of laws of evolution. The evolution laws say that applications need change to keep fulfilling user requirements, but that those changes

increase the application's complexity, and unless anti-regressive work is done, the complexity of the application will impose a limit on its ability to accommodate future change [Lehman '85]. The work of Lehman and Belady is based on the analysis of a single application. Therefore, the laws of evolution cannot be considered universal or proven. Several recent experiments on Open Source Systems (OSS) have shown that the law that indicates that software growth is linear does not apply to all applications [Godfrey '00; Capiluppi '04a; Capiluppi '04b]. With OSS it became possible to further explore Lehman and Belady's findings. The replication of their experiments has shown that, as expected, their findings are not universal. In particular, that growth trends change depending on the way in which the analyzed property is measured, and that some applications present different growth patterns. For example, the measurements chosen may change the growth trends [Capiluppi '04b], the granularity of the measurement also changes the trends of growth [Gall '97]. Therefore, depending on the case study and the metrics used there are different evolution patterns [Godfrey '00; Capiluppi '04a]. However, other works seem to confirm other Lehman's findings. For instance Capiluppi et al. found evidence of anti-regressive work (reducing the depth of the structure of modules) on Open Source Systems [Capiluppi '04a]. However, it has not been tested yet that the changes that reduce the complexity of an application make its evolution easier.

Another set of works has tried to fit the evolution of size and relations in software to a power law distribution. Power law distributions are characterized by having a long tail. An example of a power law distribution is the Pareto distribution (e.g. 20% of the population controls 80% of the wealth). The systems that fit power law distributions usually are tolerant to failure [Belle '04]. Van Belle proposed to check if the modularity of an application made it tolerant to changes. He found that the distribution of changes over time is close to a power law distribution [Belle '04]. However, it is not clear to what extent the tolerance to changes is indeed due to the modularity of the application. Herraiz et al. [Herraiz '07] found that several metrics of size and complexity followed a double Pareto distribution (i.e. a Pareto tail but a log-normal distribution body), and used the distribution to successfully predict software growth. Similarly, Wu proposes to view software evolution as a Punctuated Equilibrium, i.e. long periods of small changes interrupted occasionally by bursts of large changes [Wu '06]. In other words, software evolution is characterized by a power law distribution of the change size, measured as commit transactions and as structural changes. Finally, Myers [Myers '03] models software as directed graphs that are related by inheritance and aggregation in classes, and calls in methods. These in-

degree and out-degree distributions exhibit rough power-law scales. The long tails imply broad spectrums of reuse (in-degree) and complexity (out-degree). There are just few components with large in and out degree, i.e. with a high internal complexity and high responsibility. Most of the components are simple and highly reused (small out-degree and large in-degree), and few components are complex and used in few specific contexts (large out-degree, and small in-degree). He found that the classes that evolve most quickly tend not to interact directly with each other, i.e. they evolve separately. Moreover, he found that evolution rates are related to the collaboration graph, e.g. complex components evolved quicker than highly reused components.

2.3.3 Finding source code issues using software history

The analysis of source code history to assess the quality of applications started with the analysis of change coupling of a large and old telecommunication switching system [Gall '98]. Logical dependencies were found by identifying **co-changes** i.e. source code entities that are changed at the same time, by the same person and with the same description message. Co-changes evidence relations among source code entities: those relations include call dependencies shown in the design but also hidden relations. The authors proposed to refactor hidden relations so that the application conforms to the design.

Livshits and Zimmermann [Livshits '05] mine source code history to find frequent call patterns, i.e. methods that are usually called inside the same method. The violations to these call patterns permits the identification of possible logical errors like closing a stream without opening it.

Two studies use co-changes to identify some of the bad smells proposed by Fowler: Shotgun Surgery and Parallel Inheritance Hierarchies (see Table 2-1). Xing and Stroulia [Xing '04] identify changes at a fine grain by finding the differences between the XMI representations of two versions of an application. The co-change of these structural changes was analyzed using support[3] and confidence[4] thresholds [Xing '04]. Girba et al. [Girba '04b] also find these bad smells by tracking changes of structural metrics. They propose to find shotgun surgeries by finding classes that increased in number of statements, but kept the same number of methods

[3] Support is the minimum number of commit transactions that modified a set of source code entities [Xing '04].

[4] Confidence is the minimum ratio between the support of the pair of source code entities analyzed and the support of just one of the source code entities [Xing '04].

from one commit transaction to the next. They identify parallel inheritance hierarchies by locating classes whose number of children increases from one commit transaction to the next. Besides, they can track other structural co-changes that may indicate inappropriate abstractions like parallel increase of complexity, bugs, or semantics [Girba '04b].

The analysis of patterns of changes across source code entities can reflect source code issues, as well as logical faults in the source code.

2.3.4 Software history as a suggestion resource

Mockus and Weiss used history measurements, such as the number of modules touched, the number of developers involved, or the number of changes for developing a method, for predicting the risk of software changes [Mockus '00].

Bieman et al. [Bieman '03] identify change-prone classes and their co-change relations, and tried to see if the change behavior of classes is related with classes' characteristics like structural metrics and design patterns. They found that classes that implement design patterns are more change prone than those that do not implement design patterns, and that change prone classes tend to be larger.

Ying et al. [Ying '04] use association rule mining to recommend to the developer files that may need to be changed during modification tasks. The interestingness of their recommendations is ranked using structural relations. A recommendation is ranked as obvious, neutral or surprising depending on the structural relations between the files. Obvious recommendations occur when the modified method refers to one or more classes defined in the other file: the reference includes accessing or writing to fields, calling methods, and creating or casting objects of such classes. Neutral recommendations occur when the unmodified methods refer the other file, or when the two files have weak relations such as being in the same directory, or one file implements the interface defined by the other. Surprising recommendations are those that are not supported by any kind of structural reference.

Zimmerman et al. [Zimmermann '03] proposes a similar recommendation system at method level by mining co-changes (methods that changed in the same commit transaction), but these recommendations are not ranked.

2.4 Summary

Tackling source code issues can be identified as anti-regressive work because correcting them would not add functionality but improve the internal state of the application. Anti-regressive work is believed to improve the ease of changing the application. Given that anti-regressive work does not add functionality, the resources allocated to it are usually very limited. Therefore, it is important to maximize the impact of anti-regressive tasks.

There are two problems about the current knowledge of source code issues: firstly, there is no evidence to indicate that the proposed complex SCIs indeed exist in applications; secondly, the impact of source code issues (complex or primitive) is unknown.

An ideal way to deal with source code quality would be a conceptual framework that relates concepts like the traditional design principles to the design heuristics as well as the relation between design heuristics and source code characteristics, like complex source code issues do. However, the complex source code issues previously presented (anti-patterns and design flaws) have three main issues: they are incomplete, subjective and the concepts they provide may be insufficient to characterize real-world problems. Firstly, the approaches mentioned just cover a small fraction of the primitive source code issues and violations of source code heuristics proposed in the literature. Secondly, the rationale to compose several SCIs is debatable: different researchers have derived different models from similar sets of inelegances. Finally, although concepts like anti-patterns offer a higher level of abstraction, they may impose restrictions on what can be modeled, neglecting alternative ways in which real code problems are presented; but also, they may leave behind other forms of inelegant code.

Most of the work on SCIs has focused on their detection [Baker '95; Mayrand '96; Baxter '98; Ciupke '99; Kataoka '01; Marinescu '01; Simon '01; Emden '02; Marinescu '02; Tahvildari '02; Mens '03; Girba '04b; Marinescu '04a; Marinescu '04b; Trifu '04; Xing '04; Crespo '05; Munro '05; Trifu '05; Moha '06a; Moha '06b; Moha '08], rather than on their impact on the maintainability of the application [Lague '97; Ducasse '99; Fenton '00; Chou '01; Antoniol '02; Ratiu '04; Kim '05; Geiger '06; Li '06; Aversano '07; Krinke '07; Krinke '08]. Although all the experiments that analyze the impact of source code issues are based on software history analysis, there is no integrated methodology to analyze the impact of source code issues. The results on the impact of diverse SCIs are not comparable, because different studies analyze different aspects, or analyze the same aspect in a different way. Moreover, the analyses of the

impact of SCIs have been based on undocumented assumptions such as SCIs that are volatile or stable are harmless. Finally, some analyses on the impact of SCIs use inaccurate process to gather the data about the history of the application. The history gathering has been inaccurate in two ways: first, the identity of SCIs is not guaranteed e.g., when analyses do not make origin analysis; and second, periods are neglected ignoring intermediate changes e.g., when studies just consider releases.

In this thesis, we propose a methodology to analyze the impact of source code issues in changeability in order to prioritize their handling. Changeability was chosen because it is a core factor of maintainability. Maintainability was chosen because among all the quality attributes that a software application should have, maintainability is the one that will permit the application to survive for longer, i.e. to be a successful survivor of changes in the users' needs.

We decided to measure changeability using software history analysis because software history has been shown to be a better predictor of maintainability and changes than software structure. Software history analysis has several issues: it requires tracking historical and structural information, changes may not reflect effort, and changes cannot be isolated from other changes. Nevertheless, it has several advantages. Software history analysis does not assume that changeability depends only on structural properties. Other source properties like consistency of the language used or the quality of the documentation may also affect changeability. Moreover, software history analysis is more accurate to assess effort than other methods because it is not vulnerable to subjectivity or memory like questionnaires; additionally, the information required is widely available and can be processed automatically. Finally, software history analysis could give relevant information to propose complex SCIs that would be objective (with relations that indeed exist), and would characterize real-world problems.

Given that the analysis of source code history has shown that it is possible to find facts about the nature of programs from the analysis of several instances [Myers '03; Belle '04; Wu '06; Herraiz '07], we assume that applications are not unique and that very different applications have commonalities. Such commonalities are more likely to be caused by the characteristic analyzed than by random similarities among the applications.

Finally, we argued that origin analysis could be improved with a lighter algorithm.

Chapter 3. State of the art on clone analysis

This chapter explains why cloning was selected as case study to test the methodology proposed.

Cloning is one of the source code issues whose impact has been analyzed the most [Monden '02; Kim '05; Geiger '06; Aversano '07; Krinke '07; Krinke '08], in comparison with other SCIs [Fenton '00; Chou '01; Ratiu '04; Boogerd '08]. The high amount of studies analyzing the impact of clones might be due to the large amount and variety of tools to detect cloned code [Johnson '93; Baker '95; Davey '95; Kontogiannis '96; Mayrand '96; Baxter '98; Ducasse '99; Krinke '01; Marcus '01; Kamiya '02; Lucca '02; Komondoor '03; Calefato '04; Cordy '04; Wahler '04; Ducasse '06; Koschke '06; Li '06; Liu '06; Evans '07; Merlo '07; Roy '08; Smith '09], in comparison with tools to detect other SCIs [Ciupke '99; Kataoka '01; Marinescu '01; Simon '01; Emden '02; Marinescu '02; Tahvildari '02; Mens '03; Girba '04b; Marinescu '04a; Marinescu '04b; Trifu '04; Xing '04; Crespo '05; Munro '05; Trifu '05; Moha '06a; Moha '06b; Moha '08]. However, there is no certainty about the effect of clones [Koschke '07], maybe because most of the research on clones has focused on just one of the several claimed effects [Kim '05; Geiger '06; Aversano '07; Krinke '07], namely inconsistent changes. A wider approach of the effect of clones would produce a more comprehensive result.

Cloning is an appropriate case study to assess the usefulness of the methodology for several reasons. First, there is a lot of infrastructure available to gather data automatically about clones. Second, there is a need for a wider approach for the cloning issue. Third, there is no consensus about the effect of clones.

This chapter summarizes the technical issues that one must consider when applying the methodology to a particular SCI; that is, defining the SCI, identifying the types of SCI, detecting instances of the SCI, and tracking such instances across commit transactions. The goal of the chapter is to motivate the technical decisions to apply the methodology to clones.

3.1 Definitions

There are several definitions for clones. However, general definitions are fuzzy. Terms like near duplication [Baker '95], near miss [Baxter '98], very similar [Lague '97], indistinguishable [Antoniol '02], minor [Burd '02], substantially [Baker '95], significant [Basit '05a], mutant [Mayrand '96], etc. are common without a detailed description of what the authors mean by them.

Definitions independent of the detection algorithm define clones as the result of copying and modifying the copied code [Baker '95; Mayrand '96], or as redundant code [Ducasse '99]. However, not all cloned code comes from copy & paste [Baxter '98]. Moreover, naming cloned fragments as redundant code may imply the obligation of eliminating cloned fragments. However, there are some studies that suggest that some clones cannot be removed [Mayrand '96; Balazinska '00; Kim '04; Kim '05], and other studies that indicate that removing some clones may reduce the stability, testability or understandability of the application [Cordy '03; Kapser '06a].

The ambiguity on the definition of clones has been reduced with automatic tools to detect them. Most of the tools identify cloned fragments by describing a minimum level of similarity and a maximum level of dissimilarity. In [Baker '95] two fragments are cloned if the maximal section in which they are similar is larger than a threshold of 15 lines of code. They are similar if a global substitution of names of variables and of constants of one fragment produces the other, and the fragments differ on comments and white spaces [Baker '95]. For Baker a fragment is cloned if there is another fragment of six lines of code in the same level of nesting ignoring layout and comments [Baker '07]. Balint et al. identify two fragments as cloned if they have at least seven lines of code with identical chunks that can be interrupted by different chunks, every identical chunk should be at least three lines long, and every different chunk must have at most two lines of code [Balint '06]. Kontogiannis et al. define four scenarios for which two fragments are cloned: identical except for blank characters, identical structure (i.e. identical except for data types or variable identifiers), identical structure except for modified statements or expressions, or identical structure interrupted by added, missing or changed statements or expressions [Kontogiannis '97]. Bakota identifies two fragments as cloned if they have the same AST (Abstract Syntax Tree) node types in the same order [Bakota '07].

Therefore, most of clone definitions are dependent of the detection algorithm used, and the level

of similarity is usually omitted by providing customizable thresholds for the similarity levels. In fact, there is no agreement on the level of similarity that two fragments should have to consider them cloned; the disagreement among experts has been as high as 60% of the set of detected clones [Walenstein '04].

Clones create relations among the source code entities that share a similar same code fragment. Note that this relation is not transitive. For instance, if the method *A* shares a clone with the method *B*, and the method *B* shares a clone with the method *C*; it does not mean that the method *A* shares a clone with the method *C*. Notice that the fragment that *A* and *B* share may be different from the fragment that share *B* and *C*.

All the source code entities that share the same clone form a clone family. For instance, consider that in the previous example there is a fourth method *D* that shares the same fragment shared by the methods *A* and *B*. In this example, there would be two clone families, the one formed by the methods *A, B,* and *D*; and the one formed by the methods *B* and *C*.

3.2 Identification of clones

Before applying clone detection, it is recommended to clean the code by removing comments, and by standardizing the style of syntax, indentation, and whitespaces; this process reveals the essentials of code fragments.

Given that clones are defined as fragments of code that are similar to other fragments, it is common that the clone detection process is done in two phases, the first one to find pairs of similar fragments, and the second one to find the groups formed by the sets of pairs that are similar among themselves. Therefore, clone detection depends on two aspects: the representation of fragments of code, and the comparison function that indicates the similarly between those fragments. The quality of clone detection techniques depends on the accuracy of the code representation as a hash function, and on the relation between performance and precision that the comparison algorithms give. A summary with the most common techniques are shown from Table 3-1 to Table 3-5.

3.2.1 Advantages and disadvantages of each code representation

A code representation is good if different fragments of code have a low chance of being transformed into the same code representation.

The quality of a code representation for detecting clones also depends on factors like the awareness of syntactic units, and of their semantics (e.g. the types used, the methods called, or the variables referred). Given that some code representations do not consider the syntactic units analyzed and their semantics, their accuracy depend on the normalization of raw code before converting it. This normalization aims to clean the code by eliminating comments; standardizing the style, layout, line breaks, whitespaces, and indentation; changing literals to wild cards; eliminating irrelevant tokens; and translating identifiers to special tokens that are uniquely numbered to mark the reference of a particular identifier. This last step is necessary to locate fragments that have renamed identifiers.

Text fingerprints are lightweight code representations because they do not require knowledge of the syntax of the programming language (see examples in Table 3-1). Text fingerprints permit taking into account the semantics of the code that is revealed by the names of variables, types, and methods. However, they depend greatly of the quality of the code normalization to dismiss irrelevant differences such as layout, whitespaces, name of variables, comments, etc. Depending on the granularity level chosen to represent the code, the comparison algorithm can be more or less tolerant to differences in the cloned fragments. For instance, if the fingerprints compared represent lines of code, a single variable renamed could be identified as a difference. Finally, string representations cannot identify syntactic unit boundaries e.g. the beginning and end of declarations. Therefore, text fingerprints may produce **meaningless** clones. Meaningless clones are partial blocks of code that are not semantically related e.g. a method return followed by a method declaration, or the end of a conditional followed by the beginning of a loop.

Table 3-1. Approaches to detect clones using text fingerprints.

Code representation	Comparison/Similarity function (tool)
Normalized sub-strings of a code fragment.	Exact matching on fingerprints of LOCs. [Johnson '93] Levensthein distance on strings of characters. Each character represents an HTML or ASP tag in the file.[Lucca '02] Euclidean distance on vectors with the frequency of each HTML or ASP tag in the file [Lucca '02] Equality on LOCs allowing gaps (Duploc) [Ducasse '06]
Identifiers and comments	Latent Semantic Analysis [Marcus '01]
N-grams (normalized set of N statements)	Frequent sets [Smith '09]

Tokens are groups of characters that form an atomic element in a string. Some token representations can be found in Table 3-2. Given that tokens are a type of string representation, they share some of its disadvantages, like the dependency on the normalization of code, the unawareness of syntactic units. However, they are more resilient to different formatting. Given that they handle small strings, they can also tolerate minor differences.

Table 3-2 Approaches to detect clones using tokens.

Code representation	Comparison/Similarity function (tool)
Functors. A functor is a sequence of tokens without identifiers or literals	Suffix tree matching (Dup) [Baker '95; Baker '07]
Normalized functors.	Suffix tree matching (CCFinder) [Kamiya '02]
Sequences of the statements in a block. Each statement is the hash of the tokens that statement.	Frequent set depending on the maximum gap allowed (CP-Miner) [Li '06]

Syntax fingerprints permit a higher precision than string or token representations because they exploit the syntax of the programming language so that the comparison algorithms are aware of syntactic units, e.g. Table 3-3. This means that syntax representations do not find meaningless clones. The tolerance to differences in syntax representations depends on the capacity of the algorithm to match syntactic units of different granularity. However, note that syntax representations that depend on a complete parsing may not be very useful for analyzing source code repositories, as they would not give any answer whenever the source code stored does not compile. This means that syntax representations are likely to find clones with structural similarity, but their results are not ideal concerning textual similarities. Besides, representing code depending on the syntax of the programming language requires identifying and supporting different versions of the grammar of the programming language.

Table 3-3. Approaches to detect clones using syntax fingerprints.

Code representation	Comparison/Similarity function (tool)
Abstract Syntax Trees (AST)	Sub-tree matching. Nodes compared are chosen with a hash function. (CloneDr) [Baxter '98]
Serialized ASTs	Frequent set of statements. AST as XML. [Wahler '04]
	Suffix tree of tokens represented by the AST node. AST in pre-order. [Koschke '06]
Island grammars i.e. fragments of the grammar.	Text line comparison [Cordy '04; Roy '08]
Structural Abstractions	Tree matching, where a node can be matched to a sub-tree. AST as XML. (Asta) [Evans '07]

Metrics fingerprints are useful for lightweight and simple comparisons but this does not necessarily mean that representing code with metrics increases the performance because the calculation of the metric may be very complex. Table 3-4 has examples of metrics representations of clones. The accuracy of metrics depends on the characteristics of the code that are taken into account for representing the code. Similarly, the balance between structural and semantic similarity and the tolerance to differences depend on the metrics used.

Table 3-4. Approaches to detect clones using metrics fingerprints.

Code representation	Comparison/Similarity function (tool)
Indentation metrics	Clustering of functions using neural networks where the similarity distance is the Euclidean distance [Davey '95]
Size metrics	Visual inspection, metrics on functions with similar names [Calefato '04]
Structural & Control flow metrics	Clustering of functions using *k*-distance algorithm where *k* is a vector of thresholds for the set of metrics. Metrics per blocks. (CLAN) [Merlo '07] Three types of similarity values: equal (all metrics are equal), similar (at least one metric differs below a threshold), different (at least one metric differs above the threshold). Types of metrics (name, layout, expressions, control) used as levels of comparison. Metrics per function [Mayrand '96].
Structural, control flow, i/o, keywords metrics per statement	Markov model guides the statements to insert or delete to align two fragments, with a minimal cost. This minimal cost is the similarity between the fragments. The best match is calculated using Viterbi's algorithm. [Kontogiannis '96]

Dependencies fingerprints are useful for identifying clones in which the order of the statements may have been changed without changing their semantics, like in plagiarism analysis. An example of dependencies fingerprints can be found in Table 3-5. Dependencies

representations are costly to calculate, but their accuracy is high for semantic similarity. However, they may miss clones whose similarity is mostly structural.

Table 3-5. Approaches to detect clones using dependencies fingerprints.

Code representation	Comparison/Similarity function (tool)
Program Dependency Graphs (PDG), dependencies for control and data	K-length match (Duplix) [Krinke '01] Isomorphic sub-graphs with backwards slicing [Komondoor '03] Isomorphic sub-graph matching [Liu '06]

Mixed fingerprints allow overcoming the disadvantages of some approaches with the advantages of others. This means that mixed approaches are likely to have better precision than any other detection mechanism by itself. However, they are resource consuming, and therefore should not be used for analyzing clones over snapshots.

Table 3-6. Explanation of the approaches used to compare different representations of code.

Comparison approach	Explanation
Exact match	It is an algorithm to assess if two strings are the same or not
Levenshtein distance	It counts the edits necessary to make a fragment equal to other fragment
Euclidean distance	It is the mathematical length of the line that connects two points in an n-plane space
Lines equality, with gaps	It is similar to the exact matching but one has to decide on the maximum size of allowed differences that still permits detecting clones
Suffix trees	They are trees used to represent strings as tokens and their location in the string. Each branch of suffix tree has the common prefixes of its sub-trees, these common parts are the clones.
Intervals of metrics	They indicate the values that a metric can have to consider two fragments similar.
Markov models	It is a model in which future states are reached though a probabilistic process depending of the present state. The Markov model in [Kontogiannis '96] guides the matching process of two fragments. Fragments are begin-end blocks of statements represented as a set of metrics. The authors do not discuss the advantages and disadvantages of the approach.
Frequent sets	It finds small sets of tokens that indicate cloned code.
Latent Semantic Analysis	It finds the words that discriminate one file from the rest of files; the files that have a similar set of discriminators are placed in the same cluster. To find the discriminators, each file is represented with the identifiers and the words inside comments. Common words in the whole dataset are eliminated from the analysis.
K-distance clustering	It finds the groups of points in a space that are separated at most by a distance k, it requires a way to calculate the distance between fragments
Sub-tree matching	It finds the largest sub-trees of almost exact similarity. The algorithm in [Baxter '98] starts by reducing the number of sub-trees to compare using a hash function, i.e. it clusters the sub-trees in groups that are likely to be similar. The hash function ignore small sub-trees to permit differences in the code fragments such as renaming parameters, or gaps of added/deleted statements. The similarity between two sub-trees is twice the shared nodes over the number of nodes in both sub-trees compared.
K-length graph matching	It finds paths of a limited length (k) that have a bijective relation between the two PDGs. The path is constructed by evaluating every possible vertex that complies with the restrictions of the bijective relation, until it reaches the length required or there are no more vertices that comply with the restrictions of the bijective relation. [Krinke '01]
Isomorphic sub-graphs	It finds similar sub-graphs in two PDGs. The algorithm starts with the set of nodes that match, to construct the isomorphic sub-graphs. The sub-graphs are constructed by finding the backward and forward slices of the initial nodes. The aim of this approach is to extract the clone by reorganizing the gaps respecting data and control constraints.

Table 3-7. Comparison of the approaches used to compare different representations of code.

Comparison approach	Performance	Dissimilarity allowed	Pros-Contras (finds meaningless clones?)
Exact match	High.	None	None. (Yes)
Levenshtein distance	Computationally expensive $O(n^2)$ n = size of the fragments	Small gaps: renaming & editions	Does not permit to weight differently different changes (Yes)
Euclidean distance	Inexpensive to calculate	Any type gap	Does not consider order, so produces false positives (Yes)
Lines equality, with gaps	High	Any type gap	It is difficult to decide the maximum size of the gaps allowed (Yes)
Suffix trees	High: finds the longest common substring in linear time	Gaps introduced by replacing tokens by wildcards. Some permit parameterized gaps.	None. (Yes)
Intervals of metrics	High	Any type gap	Permits to weight the metrics. But increasing the similarity thresholds increases the false positives (No)
Frequent sets	Low	Any type gap	Does not consider order, so produces false positives (No)
Latent Semantic Analysis	Low	Any type gap Does not recognize structural similarities	It depends on the quality and consistency of the names. (No)
K-distance clustering	High	Any type gap	It is difficult to decide the minimum distance(k). Depends on the quality of the code representation used and on the distance algorithm. (Yes)
Sub-tree matching	Low	Small gaps	Depends on the order of the statements. (No)
K-length graph matching	Low. Quadratic complexity. Runs in inverse logarithmic time.	Any type gap. Finds clones with re-ordered statements.	It is difficult to decide an appropriate k. (No)
Isomorphic sub-graphs	Medium. Runs in polynomial time.	Any type gap. Finds clones with re-ordered statements.	None (No)

3.2.2 Advantages and disadvantages of each comparison approach

Table 3-6 explains each of the approaches used to compare the code representations. Moreover,

Table 3-7 evaluates each comparison algorithm in terms of time and memory performance, tolerance for different levels of similarity, and facility of use. The comparison shows that the tolerance of dissimilarity is inverse to the performance, and that the success of many comparison algorithms depends on the quality of the code representation.

3.2.3 Comparison of the most popular clone detection tools

Deciding between recall[5] and precision[6], may depend on the applications to analyze, because clone detection tools may have different recall and precision depending on the programming language analyzed [Baker '07]. Given that there is no consensus on what a false positive means in the cloning community [Walenstein '04; Kapser '07], we prefer a high recall over a high precision. A rich set of clone candidates (i.e. high recall and a large variety of clones) permits to eliminate those clones, that depending to the study, should be considered false positives.

Both speed and RAM are important for applying the methodology. RAM usage is important because it is likely that when the clone detection algorithm is called there is already a large amount of information on RAM memory maintaining information about the application and its changes. Speed is important because it is necessary to find the clones for all logical changes analyzed, therefore the performance of the application would have a great effect on the time required to collect the data.

A comparison of the most popular tools on detecting clones is presented on Table 3-8. The table summarizes the results of the study done by Bellon et al. to evaluate several aspects of clone detection tools. Given the aspects analyzed, the study [Bellon '07] is appropriate to evaluate technical advantages of the tools for applying the methodology. As the table shows, CCFinder is the tool that offers the most advantages. However, given that it is a token-based detection tool, its precision is low and it tends to find meaningless clones.

[5] Recall is the amount of clones found by the detection tool that are correct, over the total number of clones in the application analyzed.

[6] Precision is the amount of clones found by the detection tool that are correct, over the total number of clones found by the detection tool.

Table 3-8. Comparison of the main clone detection tools [Bellon '07]. A check means that the tool is better in that characteristic, in comparison to the other tools.

Tool	Recall	Precision	Richness of output[7]	Speed	RAM	Total ✓
Dup [Baker '07]	✓			✓	✓	3
CCFinder [Kamiya '02]	✓		✓	✓	✓	4
Duploc [Ducasse '06]	✓		✓		✓	3
CloneDr [Baxter '98]		✓				1
CLAN [Merlo '07]		✓	✓	✓		3

3.3 Classifications of clones

We have found that there are four ways to classify clones: according to characteristics of the cloned fragment (that we call characteristics C1), according to characteristics of the relation between the cloned fragment and the method (that we call characteristics C2), and according to characteristics of the clone family (that we call characteristics C3). The characteristics of the clone family can be are shared by all fragments in the family (that we call characteristics C3.1) or not (that we call characteristics C3.2). Each type of clone classification presents a set of characteristics that exemplify it.

The rest of the section explains for each type of classification, the characteristics used in empirical studies, the ways used to calculate the characteristic, and the empirical results obtained (summarized on Figure 5-8).

3.3.1 Classification of clones by attributes of the clone fragment (C1)

The only characteristic of the clone fragment that has been used to classify clones is the lifetime.

Lifetime: The lifetime of a cloned fragment is volatile if it is deleted within eight commit transactions or less of its creation, otherwise it is persistent [Kim '05]. An empirical study on two projects showed that most of the clones were volatile, and that persistent clones were difficult to refactor [Kim '05].

3.3.2 Classification of clones by relation between the clone and the method

[7] Types of clones that the application is capable of detecting (see Table 3-9). The higher the amount of types of clones, the richer the output is.

(C2)

Only two characteristic defined by the relation between cloned fragment and the source code entity that holds it has been used to classify clones: the age of the method and the percentage of the method affected with clones.

Age: Monden et al. showed that the older the code is the lower its percentage of clones: 18% of young code (less than 4K days) was cloned, 13% of middle-aged code (between 4K and 8K days) was cloned, and 11% of old code (more than 8K days) was cloned [Monden '02].

Percentage affected: Several authors have measured the percentage of the application affected by cloning [Baxter '98; Ducasse '99; Antoniol '02; Kamiya '02; Monden '02; Ueda '02; Kapser '04; Al-Ekram '05; Basit '05a; Geiger '06; Li '06]. They have found that cloning in an application is usually below 25% of the source code entities analyzed [Mayrand '96; Ducasse '99; Monden '02; Li '06]. They also found that the percentage of cloning varies across different subsystems of an application [Baxter '98; Antoniol '02; Li '06]. They could not find any relation between the percentage cloned and the number of co-changes of clone-related files [Geiger '06], and neither any regularity among the percentage cloned in applications of the same domain [Al-Ekram '05], except for operating systems [Antoniol '02].

3.3.3 Classification of clones by relations shared among cloned fragments of the same family (C3.1)

Given that clones are defined in terms of the relations among cloned fragments, there are several characteristics shared by cloned fragments of the same family that have been used to classify clones: similarity, scope, size of the fragment, size of the family, intention, distance, percentage of literals, and number of tokens eliminated in case the clone family is refactored.

Similarity levels: The levels of similarity among cloned fragments of the same families are summarized in Table 3-9. All authors found that the lower is the level of similarity; the higher is the number of cloned fragments in the family. For instance, Baker found that parameterized clones inside a file are more common than exact clones [Baker '95]. Balazinska et al. found that the majority of clone families have several differences (i.e. more than 50) [Balazinska '00]. Maryland et al. classified the number of clone relations found per type of clone, and they found that as similarity decreases the number of clone relations increase [Mayrand '96]. Marcus and

Maletic showed that the semantic similarity decreases as the number of files involved in the clone family increases [Marcus '01]. Li et al. depicted the percentage of lines cloned versus the minimum threshold for clones, and they found that the lower the minimum size of the clone is, the higher percentage of code cloned [Li '06].

Table 3-9.Levels of similarity in clones of the same family

Level of similarity	Synonyms I [Davey '95]	Synonyms II [Baker '95], [Baxter '98],[Kamiya '02]	Synonyms III [Mayrand '96]	Synonyms IV [Balazinska '00]
Exact	Type I	Exact	Same name, layout, expressions, and control flow	Same names, types, and context
Renames, small addition/deletions	Type II	Parameterized, Gapped	Distinct name, similar layout	Distinct names, same types, and context
Larger editions	Type III	Gapped, Near-Miss	Different layout,	Distinct names, and types, and same context
Same behavior, regardless of the structure	Type IV	Wide miss clones	Different expressions, and control flow	Distinct names, same types, and context

Scope: In general, the results concerning scope defined by a clone relation are contradictory. All authors have found that the common categories on the scope of clone relations are inside functions, and inside blocks [Kapser '03; Kapser '04; Kim '04; Li '06]. In [Kapser '03] the authors showed that most of the clones are blocks of code across different functions (40%), the second most common is Function-to-Function clones (35%). In [Kapser '04] the most common scope for clone relations is Function-to-Function (99%), and from the categories inside this scope, the most common scopes are Function (47%-48%) and Blocks (25%-42%). Li et al. report two categories for the scope of clones: by functions and by blocks. They found that the percentage of cloned fragments at function level was between 11% and 19%; while the percentage of cloned fragments at block level was between 3% and 9% [Li '06]. However, it is not clear what scope had the rest of cloned fragments.

Intention: Kapser and Godfrey [Kapser '06a; Kapser '08] propose to classify clones by the intention of developers when cloning. Forking occurs when the developer does not want to change a part of the application, but wants to enlarge the environment in which it can be used.

Templating occurs when clones are caused by common functionality or common programming style, such as API usage, boilerplate solutions, idioms, and parameterized clones. Customizing occurs when the clone requires changes depending on the environment where it is pasted; they are considered good-clones because the clone might be easier to understand than an abstraction. Finally, exact match clones usually implement crosscutting concerns and control flow checks. The ways to calculate the intention for creating the clone family, and the empirical results obtained are summarized in Table 3-10. Notice that the results may not be exact because the empirical results were obtained by manual inspection of a subset of the clones of the applications found with CCFinder. The subset of clones analyzed were obtained randomly.

Table 3-10. Types of clones by intention of the developer when creating the clone family [Kapser '06a; Kapser '08].

How to calculate it		Typical	Atypical
Forking, Templating, Customize, Exact Match		Templating	Forking
	Forking types: Hardware, Platform, or Experimental variation	Platform variation	Hardware variation
	Templating types: Boiler-plating, API, Idioms, Parameterized	Parameterized	Idioms
	Customize types: Replicate and Specialize, Bug Workarounds	Replicate and Specialize	Bug Workarounds
	Exact Match types: Cross-cutting, Verbatim snippets	Verbatim snippets	Cross-cutting

Size of the fragment: Several researchers have found that the amount of clones found is inversely related to the minimum threshold to consider a fragment a clone. For instance, several authors found that the larger the clone size the lower its number of occurrences, i.e. typical clones are small [Baxter '98; Balazinska '00; Burd '02; Li '06; Baker '07]. Others found that the typical values of size per fragment are close to the minimum threshold [Basit '05b; Kapser '08]. Probably, the typical cloned fragment size varies because of the way in which the size is calculated. For instance, the minimum threshold to consider a fragment cloned varies significantly among studies 30 LOCs [Baker '07], between 15 and 23 LOCs [Burd '02], between 3 and 16 [Li '06], and below 30 LOCs. Nevertheless, other differences among different studies such as the programming language, the domain of the application, or the clone detection tool may also explain the lack of convergent results.

Size of the family: Several researchers have found that the frequency of classes of a given size is inversely related to the size of the classes. That means that most of the clone families are

small [Johnson '94; Mayrand '96; Marcus '01; Kapser '06b; Li '06].

Distance: Studies also coincide in pointing out that cloning occurs in areas of code that are close to each other, either in terms of the hierarchy of directories [Kapser '03; Kapser '04; Kapser '06b] or in terms of the hierarchy of classes [Golomingi '01; Jarzabek '06]. The categories most common were inside the same file, of inside the same directory directories [Kapser '03; Kapser '04; Kapser '06b], or inside the same class or inside sibling classes [Golomingi '01]. The most uncommon category was across different directories [Kapser '03; Kapser '04; Kapser '06b], and cloning in different class hierarchies or cloning with an ancestor [Golomingi '01].

Wildcard tokens: Empirical studies also signal the importance of expressing gaps in clones. Many of them indicate the increase of clones detected thanks to wildcard areas. For instance, Li et al. found that the number of identifiers renamed in most similar fragments on each clone family was either zero renames or more than 5 identifiers renamed [Li '06]. In fact, between 59% and 76% of cloned fragments require renaming of identifiers. Other authors have found that constants affect less than 50% of the clone relations in the application (between 25% and 36% [Ducasse '06], and between 40% and 50% [Kamiya '02]). However there is no consensus about function names: while some authors consider that they should be wildcards [Ducasse '06], others indicate that differences in the name of functions called affect the semantics of the clone and therefore if used as wildcards increase the false positives [Kapser '08].

3.3.4 Classification of clones by relations not shared among cloned fragments of the same family (T3.2)

Although cloned families share several characteristics that can help to understand the nature of cloning, the characteristics that differentiate cloned fragments of the same family may help to analyze its behavior in comparison with other fragments in the family. The characteristics not shared among members of a clone family that have been used to classify clones are similarity to the clone, and the method call similarity.

Similarity to clone: Kapser and Godfrey propose to measure the similarity in cloned fragments located in logical structures to eliminate false positives [Kapser '06b]. They count the percentage of lines of code in which the cloned fragments differ, and eliminated those in which the difference is above 50%. Using this heuristic, they eliminated from 3% to 37% of the clone

relations found. Jiang et al. [Jiang '07] define three inconsistencies among cloned fragments of the same family: different type of tokens, different conditions in the cloned fragment, and different number of unique identifiers. All types of inconsistencies found were less than 6% of the clone families.

Method call similarity: Finally, Kapser and Godfrey propose to measure the similarity in method calls to eliminate false positives [Kapser '06b]. Using this heuristic, they eliminated from 8% to 52% of the clones found by CCFinder.

3.4 Analysis of clone history

Once the clones have been detected, it is necessary to track them over different snapshots. Clone tracking over time has been done at the level of clone families. The literature [Kim '05; Aversano '07] describe the changes that can occur to a clone family in a logical change. There are six possible cases (depicted in Figure 3-1):

- Same: None of the fragments that compose the clone family change nor does the composition of the clone family.
- Add: The clone family has new cloned fragments.
- Subtract: The clone family looses cloned fragments.
- Shift: Cloned fragments in the family change but remain similar to their previous version.
- Consistent change: All the fragments in the clone family change in the same way, at the same logical change.
- Inconsistent change: There are fragments that do not change in the same way, at the same time:
- Late propagation: If the fragment reverts to consistency in a later logical change.
- Independent evolution: If the fragment changes in a different way.

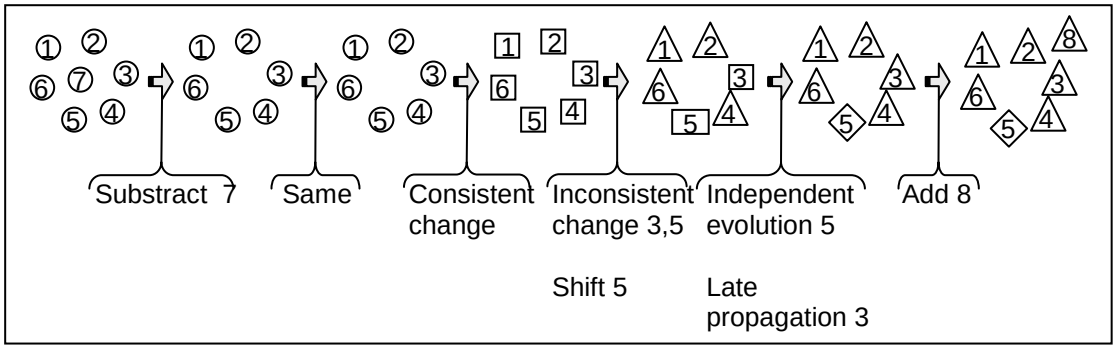


Figure 3-1. Evolution of clone families. Each element corresponds to a cloned fragment of the family, and can be identified by the number inside. Changes in the shape indicate changes in the fragment. Similar shapes correspond to similarity of the fragments.

In order to track fragments across logical changes it is necessary to compare the text and location of the fragments. There are three cases to decide if one fragment is the new version of another fragment: if their location and their text are exact; if their location is exact and their text is very similar; and if their text is exact and their location is very similar. This approach requires storing the text and location of all fragments, which may result in performance problems due to the size of the search space. Besides, storing just the fragments and their changes may not be sufficient to track clone families because whenever there is an inconsistent change or a shift, the fragment may stop sharing the commonalities of the clone family. Nevertheless, storing just the snippet shared by the family, and the location of the cloned fragments would not be enough to track all changes shown in Figure 3-1. From the logical change in which a family has fragments with inconsistent changes, those fragments stop belonging to the family, and therefore it would be impossible to differentiate how many of those inconsistent changes are late propagations, and how many independent evolutions. Differentiating between late propagations and independent evolutions is a key issue when analyzing clones because they indicate to what extent the lack of clone management causes bugs (which are fixed by late propagations).

Furthermore, the subtraction of fragments from its clone family may happen by chance in two cases. If changes in a cloned fragment result in the fragment not being sufficiently similar to the rest of the fragments in the family, it would be identified as deleted. If the method that has the cloned fragment is deleted, the fragment would be identified as an element removed from the cloned family. If the removal of cloned fragments occurs in a clone family with only two fragments, the clone family would be identified as deleted. This means that with the current approach for tracking clones over time not all cloned fragments or clone families removed mean

a refactoring of the code cloned.

Some of these observations coincide with the issues identified in the literature for tracking clones over time [Smith '09]. First, it is possible that the current model of the evolution of clone classes over time (shown in Figure 3-1) is incomplete; and therefore, it could benefit from empirical studies on the evolution of clone families. Second, there is a need for detecting and tracking cloned fragments that were not consistently changed (i.e. do not belong anymore to their initial clone family). Third, there is no definition for the accuracy of clone tracking. Fourth, there is no technique to distinguish automatically late propagations from independent evolutions.

Other authors identify some patterns of evolution of clone families as clone smells [Bakota '07]. The patterns proposed are: vanished clone instances (subtract), occurring clone instances (add), moving clone instance (independent evolution), migrating clone instance (late propagation) [Bakota '07]. For each pattern there is an alternative to detect it [Bakota '07]. However, the results had a high amount of false positives, mostly because of irrelevant clones, compilation issues, and defective evolution mapping. When analyzing only the results without false positives, the authors found that the patterns proposed indicate modification in the cloned fragments that increase the differences with the clone family. In other words, the patterns proposed indicate late propagations, cloned fragments deleted, inconsistent changes, bugs due to inconsistent changes, and cloned fragments introduced due to bug fixes. It is not clear which model for tracking clones produces more intuitive results the one based on clone families [Kim '05; Aversano '07] or the one based on cloned fragments [Bakota '07].

3.5 Summary

There is no consensus about the definition of clones. Although clones are meant for duplicated code that requires duplicated maintenance, it is not clear how to recognize if a fragment is the duplicate of another fragment. Most authors distinguish clones using a minimum similarity threshold. Nevertheless, some authors claim that it is also necessary to establish a maximum threshold of dissimilarities allowed. In any case, there are no standard thresholds.

The automatic identification of clones depends on two factors: the way in which source code is represented, and the way in which the similarity is calculated. The clone detection algorithm tends to weight more than the code representation on the precision of the output. For instance,

algorithms based on Abstract Syntax Trees originate a minimum amount of false positives. When the code representation takes into account the semantics of the source code syntax it can help to avoid meaningless clones i.e. those clones that do not represent a fragment of functionality, for instance the return of a method followed by the declaration of a new method. Nevertheless, the representations that provide the best precision rates, are also those that have a low recall, leaving aside from the analysis many potential clones. In that sense, the code representations that provide the richest variety of potential clones are those that are closer to the original source code.

In any case, the overall success of a clone detection algorithm depends on how well the code similarity algorithm complements the code representation weaknesses and vice-versa. For instance, although text representations tend to produce false positives they can be reduced significantly if the similarity algorithm parameterizes literals and names of variables, dismisses the code style, and takes into account the boundaries of source code entities. These strategies make of CCFinder one of the favorite tools for clone detection. It does not only offer a good performance thanks mostly to suffix trees, but also, thanks to the strategies to reduce false positives it can locate a high variety of clones.

The identification of clones is not enough in order to analyze their history. It is necessary to track cloned fragments and clones families over time. In particular, for cloned fragments having inconsistent changes or an independent evolution from their family. This is because any inconsistent change or independent evolution is a potential late propagation.

The next chapter introduces the methodology to analyze the impact of SCIs in SCEs.

Chapter 4. SuspiSCIUS methodology

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"

Sherlock Holmes in The Adventure of the Blanched Soldier of Sir Arthur Conan Doyle

Tackling source code issues is part of the anti-regressive work done to keep the complexity of the application manageable. It is important to be able to prioritize the impact of diverse source code issues on the ease of change of the source code entities where they occur. However, as the Background for the methodology points out (see Chapter 2), current knowledge of the impact of Source Code Issues (SCIs) is fragmented and difficult to compare. Such fragmentation obscures the effect of SCIs on changeability.

Analyzing the effect of source code issues in the source code entities where they occur requires a structured and systematic research strategy. Having a common approach for source code issues would allow to compare their impact, and therefore to prioritize it. A **methodology** is a systematic manner to accomplish tasks using a set of techniques and a set of rules to apply such techniques [Nance '88]. Methodologies guide the definition of objectives, the organization of tasks, the techniques to achieve those tasks, and prescribe rules to decide which techniques lead to the objectives fixed for the tasks [Nance '88]. A methodology for evaluating the impact of source code issues on source code entities is needed in order to provide in-depth understanding of said impact and to compare the impact of diverse source code issues.

Intermediate versions of this methodology have been published in [Lozano '06; Lozano '07b; Lozano '08a]. However, this version should be considered as the definitive one.

This chapter explains our methodology, called suspiSCIUS, and the rationale for

its phases.

4.1 The suspiSCIUS methodology

This methodology assesses the impact of a SCI on changeability of the Source Code Entity (SCE) where it is placed. The results of applying our methodology to different SCIs provide comparable indicators of their impact, and therefore act as priority indicator for their treatment. The methodology is called **suspiSCIUS** for two reasons: first, because suspiSCIUS is a mnemonic name for Source Code Issue Under Study (**SCIUS**); and second because the methodology is based on elimination of hypotheses, which is similar to finding who is guilty of a crime by eliminating suspects.

The methodology is defined in an abstract manner, that is, in terms of SCI, SCIUS, SCE, etc. Defining the methodology in abstract permits the researcher to customize it depending on the SCI of interest, and to the granularity of SCE in which the SCI occurs. The advantage of having the same methodology for diverse SCIs is that the results are comparable. To achieve that, the methodology requires adaptations for certain phases.

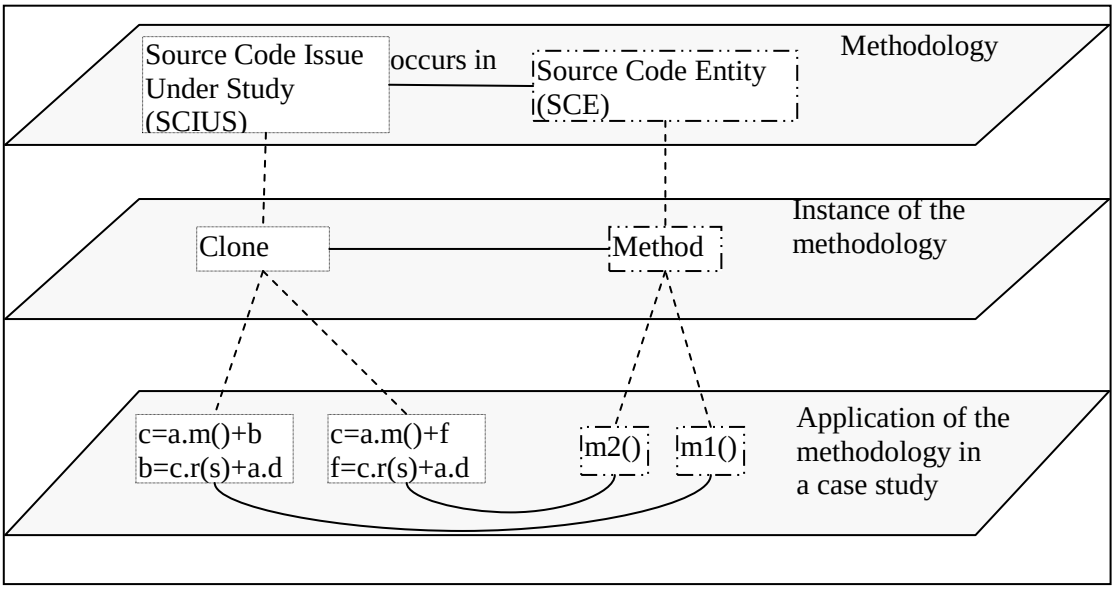


Figure 4-1. Levels of abstraction of the methodology concepts

Figure 4-1 shows a possible adaptation of our methodology for each of the concepts used at each level of abstraction of the methodology. The methodology level defines structural elements of the methodology, e.g. SCIUS, and SCE. The adaptation of the methodology instantiates the

concepts of the methodology, e.g. to clones (the SCIUS) inside methods (the SCE). The application of the methodology instance provides the actual subjects to investigate, e.g. the cloned fragment inside method *m1()*, and the clone fragment inside method *m2()* form a clone.

4.2 Concepts of the methodology

The concepts on which the methodology relies stem from the relations between SCEs, SCIUS, and changes. The purpose of the methodology is to assess the effect of SCIUS on the changeability of an application, but more specifically, on the SCEs where the SCIUS are located. Given that changeability is the ease to inject changes in an application, a key concept of the methodology is the changes that an application undergoes over its lifetime.

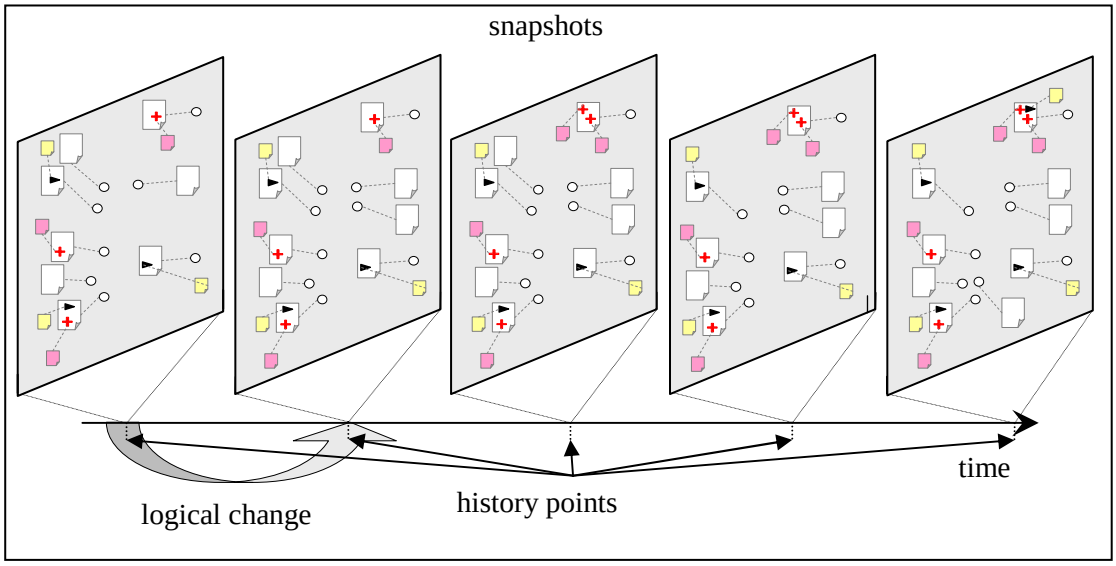


Figure 4-2. Relevant information to apply the methodology.

Concepts used in the methodology are shown in Figure 4-2. The gray vertical layers in the figure represent the status of the application at certain moments in time on the history of the application, a.k.a. snapshots. Logical changes generate snapshots. The moments in time in which a logical change is introduced in the application are called history points. In the figure, the circles inside the snapshots are the parts that compose the application at that history point, a.k.a. SCEs. The notes besides each SCE are the values of the characteristics of that entity at that moment in time. The SCIs are characteristics particularly interesting for the methodology. In the example there are two SCIs tracked. Each type of SCI has a different icon; one of them is a black triangle and the other a red cross. Each SCI has as well its own characteristics. The

characteristics of the SCI depicted with a triangle are in a smaller note in yellow, and the characteristics of the SCIs depicted with a cross are in a smaller note in pink.

The key concepts of the methodology are shown in Table 4-1. These basic concepts are used across all the phases of the methodology, in particular, to define other concepts. The rest of this section gives a detailed definition of each one of these basic concepts. Appendix A presents formulas that structure the concepts explained in this section. These concepts are used in chapters 6, 9 and 10 to explain other concepts of the methodology.

Table 4-1. Key concepts in the methodology

Key concept	Related key concepts
history of an application	logical change snapshot
SCE	characteristics of the SCE changes of the SCE SCIUS in the SCE other SCIs in the SCE
SCIUS	characteristics of the SCIUS changes of the SCIUS
Other SCIs	changes of the SCI

Source Code Entities (SCE) are the building units of an application. Most of the analysis and results that the methodology produce are at the level of SCEs. Each SCE is composed of one or several SCEs of the same or of lower granularity. For example, in Java, packages are composed of other packages or classes, classes being of a lower level of granularity than packages. Depending on the SCIUS, the pertinent granularity of the SCEs to be analyzed may vary. For instance, for analyzing god classes it is enough to have information of the properties at class level, while for analyzing god methods it is necessary to have information of the properties at method level.

Level is the granularity of a SCE. Levels can be tokens, lines of code, methods, classes, files, modules, etc.

Snapshot is the set of entities that compose the application after a commit transaction. A snapshot describes the status of the application, in terms of the entities that compose it at a moment in time.

History of an application: is sequence of snapshots ordered chronologically by commit

transaction.

Logical change is a synonym for commit transactions. As explained before a commit transaction is a set of physical changes done with the same purpose, at the same time by a single developer (see section 2.3.1). A logical change contains the set of entities that changed from one snapshot to the next snapshot in the history of the application. We use the term *changes* to refer to logical changes, and a logical change is composed of several physical changes in diverse entities of the application.

Index is an integer that indicates the chronological order of a commit transaction. This means that the index is the point in history in which a logical change occurs

Characteristic is a property of an entity at a moment in time. A particular entity has several attributes at a given moment in time. The attributes of a source code entity may include its name, size, if it has a SCI or not, if it was modified or not, etc.

HasSCIUS is a characteristic of the SCE that says if the SCE has the SCIUS or not at a moment in time.

Having defined the basic concepts of the methodology, we can proceed to explain the phases that compose the methodology.

4.3 Context and analysis phases

This methodology helps to find precise information about the impact that the SCIUS may have. Impact is considered in two ways: the impact on the changeability of the SCE where it is placed, and the impact on the changeability of the application.

Given that it is desirable to compare the impact of diverse SCIs to prioritize maintenance tasks, the methodology is generic enough to be applied on diverse SCIUS or SCEs. Moreover, the methodology can take into account results from empirical studies reported in the literature, either to exclude parts of the analysis to avoid redundancy or to include such results in the considerations taken for the set-up and the analysis. Therefore, both technical and analytic considerations regarding SCEs or SCIUS are customizable items in the methodology.

The methodology is composed by a sequence of two kinds of phases. The context phases aim to provide structured information that explains the hypotheses or that supports arguments when examining results of the analysis phases. The analysis phases aim to refute hypotheses using the

information gathered in the context phases. Context phases are derived from common data requirements of different analysis phases. Therefore, which context phases are required depends on the analysis phases that interest the user of the methodology. The methodology can be tailored by extracting only the context information required for each phase, as Figure 4-3 shows.

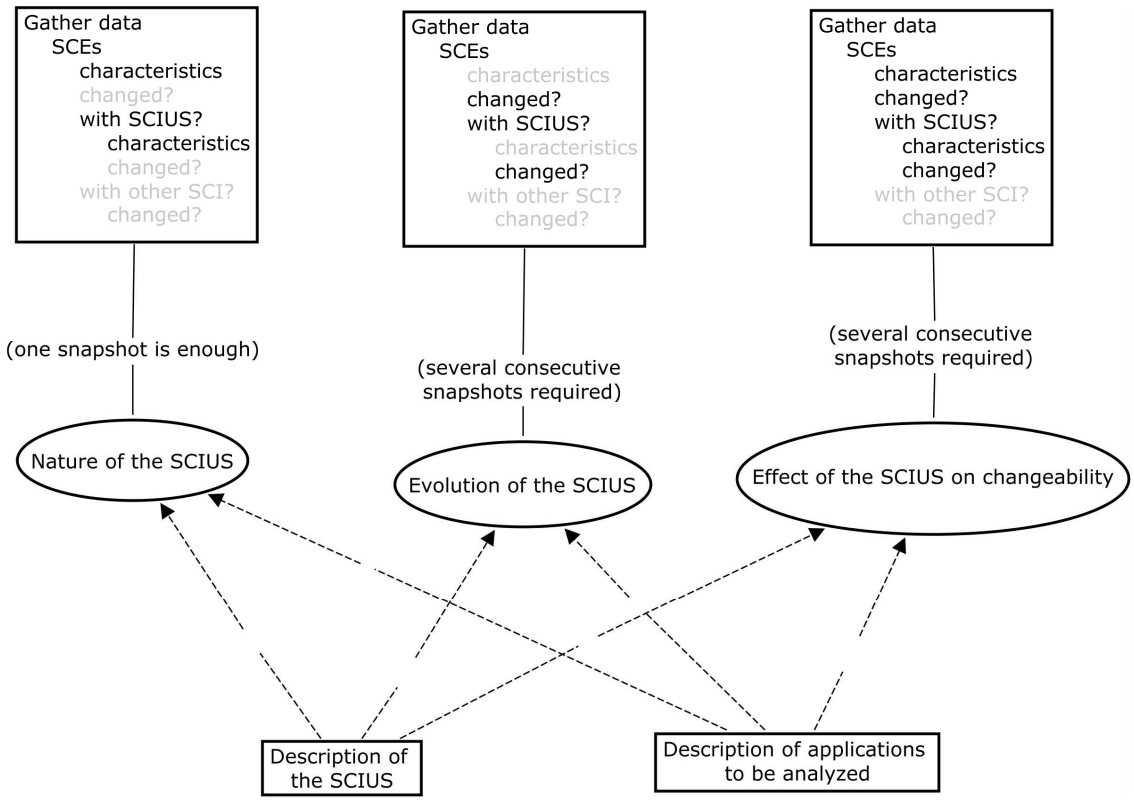


Figure 4-3. Context phases (in rectangles), and analysis phases (in circles). Depending on the analysis phase to execute, some data is not required to be gathered (in gray).

The phases of the methodology work as layers, as Figure 4-4 shows. The outer phases provide general results. The methodology should be applied from outer to inner layers, by refining the knowledge about the impact of the SCIUS. Context phases are shown as square layers, while analysis phases are shown as circular layers. The names of all phases describe the goal of the phase: context phases are named with verbs and nouns, while analysis phases are named with nouns related to the SCIUS.

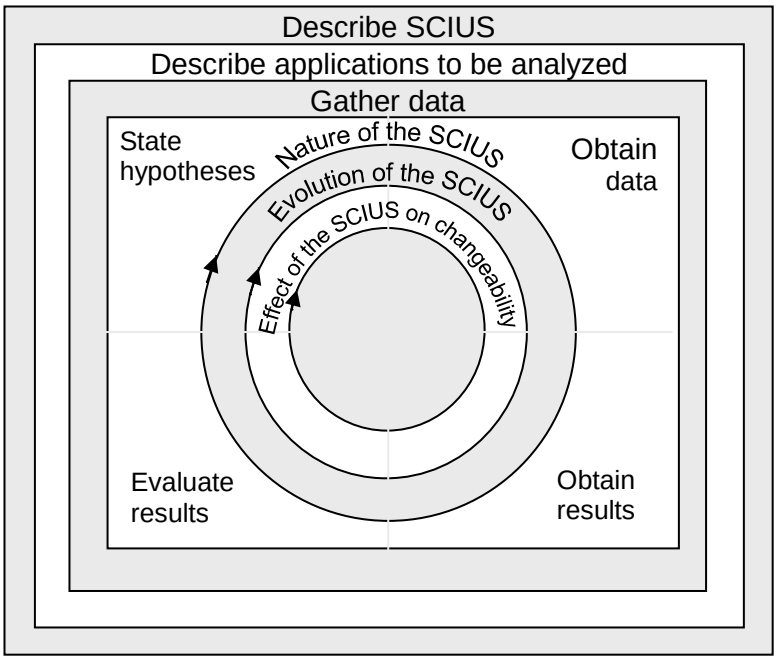


Figure 4-4. Methodology phases. In bold there are the outcomes, in italics are the steps of analysis phases.

To complete an analysis phase (circles) it is necessary to do four steps. Each step indicates the type of tasks to perform at the beginning, development, analysis, and close-up of the phase. The first step is called *state the hypotheses*: it aims to predict the results using information from related work or from previous phases. The second step is called *obtain data*: its purpose is to process raw data into information that can be related to the hypotheses. The third step is called *obtain results*, which uses statistical and graphical tests to study the veracity of the hypotheses. Finally, the fourth step is called *evaluate results* because it summarizes for which hypotheses there is rejecting evidence, and for which hypotheses there is supporting evidence. The outcome of this last step can be used to feed the first step of the following phase.

The research questions tackled by each analysis phase are described below. Phases are identified by their name in bold and italics. For each phase, there is the set of questions that it aims to answer. Given that there are phases whose research questions are too broad to tackle at once, the phases have been divided into more manageable research objectives. Each part of a phase is called a sub-phase and it is indicated with an arrow (→).

Describe the SCIUS

- Why is it worth studying the impact of the SCIUS?

→ ***Definition of the SCIUS***

- What does it mean to have the SCIUS?
- What types of SCIUS exist?

→ ***Causes and consequences***

- Why do programmers introduce the SCIUS?
- Why is the SCIUS considered harmful?

Describe applications to be analyzed

- Which application characteristics may explain commonalities or differences in the results?

Gather data

- How to obtain reliable data to evaluate the effect of SCI on the changeability of the SCEs where they occur?
- How to identify the logical changes of an application?
- How to identify the SCEs that compose the application at any moment in time?
- How to track the SCEs that compose the application over time?
- How to identify the SCIUS that affect the SCEs of the application at any moment in time?
- How to track the SCIUS at the SCEs that compose the application over time?
- How to identify which SCEs and which SCIUS changed at each logical change?
- How to characterize SCEs and SCIUS over time?

Nature of the SCIUS

- What characterizes a typical exemplar of the SCIUS?
- What do instances of the SCIUS have in common?

Evolution of the SCIUS

- Does effect of the SCIUS on the application increase over time?

→ ***Extension***

- How much of the application is affected by the SCIUS?
- How does the presence of the SCIUS in the application change over time?
- How much of the SCEs is affected by the SCIUS?
- How does the presence of the SCIUS in the SCEs change over time?

→***Persistence***

- How much of the SCE's lifetime is affected by the SCIUS?
- How does the amount of SCEs lifetime affected by the SCIUS change over time?

→***Stability***

- How many of the changes of a SCE occur when it is affected by the SCIUS?
- How does the amount of changes that a SCE may have when it is affected by the SCIUS change over time?
- How many of the changes of a SCE occur within the SCIUS?
- How does the amount of changes that a SCE has inside the SCIUS change over time?

Effect of the SCIUS on changeability

- In which cases is the SCIUS harmful for the changeability the application?

→ ***Measurement of changeability***

- How to assess the effect of the SCIUS in the changeability of the SCE where it is located?

→ ***Comparison of changeability***

- Does the presence of the SCI have any impact on the changeability of the application?
- Is the impact positive or negative?
- What is the extension of such impact?

→ ***Identification of characteristics that affect changeability***

- Which characteristics of the SCE may mislead the effect of the SCIUS on the changeability of the SCE where it is located?
- Which characteristics of the SCIUS may increase its effect on the changeability of the SCE where it is located?
- Which combinations of characteristics of the SCIUS and of the SCEs may increase the effect of the SCIUS on the changeability of the SCE where it is located?

→ ***Classification of SCEs by changeability measurements***

- What characterizes those SCIUS that severely affect the changeability of the SCE?
- What characterizes those SCIUS that do not affect at all the changeability of the SCE?

4.4 Summary

This chapter presented an overview of the methodology. The methodology is based on abstract concepts, which work as extension points to make the methodology applicable to diverse SCIs at different SCEs. The chapter explained basic concepts of the methodology and the research questions it can tackle.

The following chapter explains the context phases of the methodology. The description of each phase will describe its motivation, deliverables, steps required to produce the deliverables, and adaptation required for the SCE, and the SCIUS chosen.

Handling source code issues is part of the anti-regressive work that allows controlling the complexity of an application and therefore permits the application to evolve for longer. The methodology proposed is motivated by the need for a standard way of measurement for the impact of SCIs, to prioritize their treatment. The methodology is based on two assumptions: first, that it is possible to find facts about the nature of source code changeability from the analysis of several applications; and second, that changes of SCEs without the SCIUS represent typical changes of the SCE. The methodology is based on the generation of hypotheses, and their progressive elimination by analysis of the information collected.

Chapter 5. Source Code Issue Under Study

This chapter describes the first phase of the methodology. The goal of this phase is to establish current knowledge about the SCIUS. It is recommended to perform this phase before the following phases, to establish the research questions that have been already tackled, especially, to establish precisely the gaps in the knowledge about the SCIUS. Apart from giving hints on which phases of the methodology should be applied, this phase would contribute to establish definitions of key concepts, generate hypotheses, and predict results.

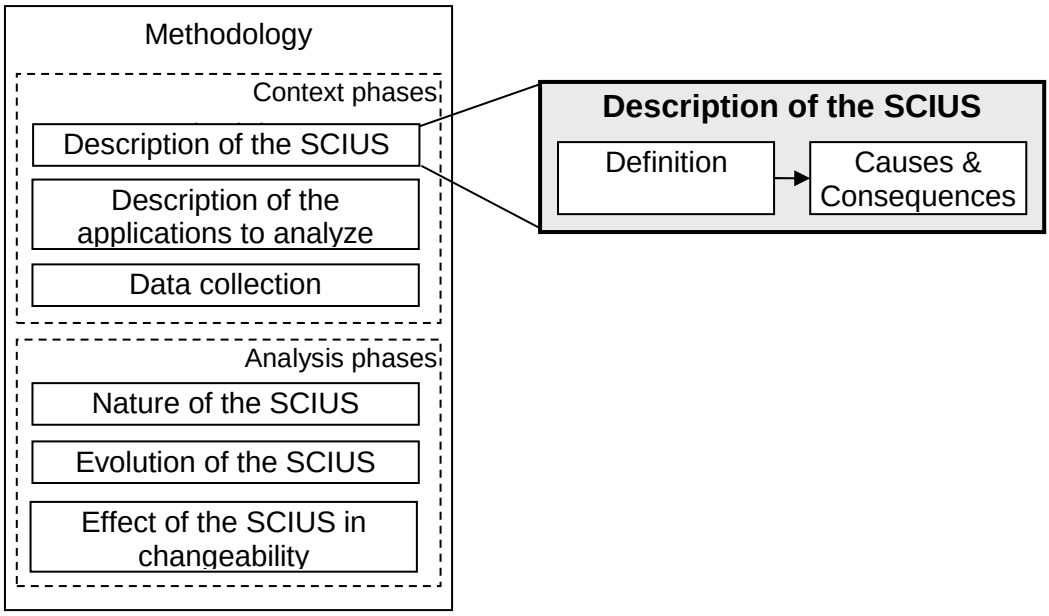


Figure 5-1. Description of the phase "Description of the SCIUS"

5.1 Phase description

This section presents the steps required to perform this phase. As Figure 5-1 shows, this phase is composed of two sub-phases: the definition of the SCIUS, and the description of its causes and consequences. Each of the sub-phases is explained in the sub-sections below.

5.1.1 Definition of the SCIUS

Figure 5-2 shows the steps required to complete this sub-phase, i.e. stating alternative definitions, and describing the types of SCIUS.

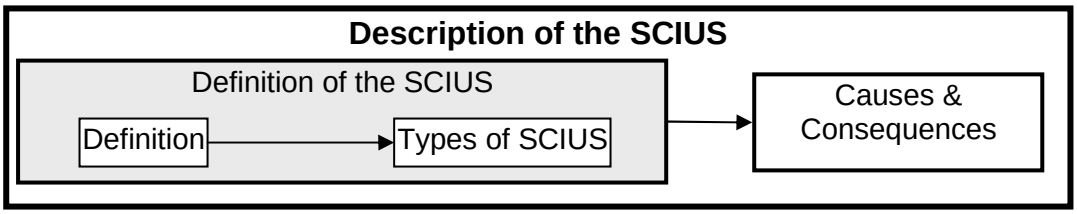


Figure 5-2. Definition of the sub-phase "Definition of the SCIUS" of the phase "Description of the SCIUS"

Deliverable: The goal of this sub-phase is to obtain a unique definition of the SCIUS to use all through the application of the methodology. This definition will be used to identify the instances of the SCIUS when applying the methodology.

Rationale: It is strongly recommended to establish a definition for the SCIUS that applies to the rest of the methodology, because defining terms serves as a framework of conventions for comparability, it makes the assumptions explicit, it says how to count the data and how to categorize it. The accuracy of the definitions helps to interpret the results, and to assess their generalizability.

Procedure: State the definitions of the SCIUS that will be used using the concepts introduced in Section 4.2 (i.e. in terms of SCEs and properties of those SCEs). Give an example of that definition. Identify the type of source code issue: if it can be attached only to a single source code entity it is entity-type; on the contrary, if it defines relations across different source code entities it is relational-type. In case the SCIUS defines relations among other instances of the SCIUS or among SCEs, state the SCEs related, their associations, and the roles of each SCE.

5.1.1.1 Nature of the SCIUS

The SCIUS have several characteristics. Characteristics can depend on the SCIUS, on the SCE where the SCIUS is placed, and on the relation between the SCIUS and the SCE where the SCIUS is placed. Typical values for each characteristic can help to describe the average instance of the SCIUS. In contrast, atypical values for each characteristic can help to describe unusual instances of the SCIUS.

Deliverable: The typical and atypical values per category, based on previous empirical results. See example in Figure 5-3.

SCIUS type by	SCIUS characteristic		Size	Typical (40) Atypical (>100)
			Complexity	Typical (simple) Atypical (complex)
	characteristic of relation SCIUS-SCE		Location	Typical(method) Atypical (class)
			Lifetime affected	Typical (volatile) Atypical (persistent)
	characteristic of relation SCIUS-SCIUS	Shared	Scope	Typical (1-3) Atypical(>10)
		Not shared	Role	Typical (copy) Atypical (seed)

Figure 5-3. The nature of the SCIUS, can be described by the typical values of each one of its characteristics. Example of the nature of a fictitious SCIUS.

Rationale: The purpose of this step is to describe the majority of exemplars of the SCIUS. Knowing the typical and atypical values for characteristics of the SCIUS may help to identify those SCIUS that should be tackled with priority (because of their high harmfulness). Moreover, some characteristics may reveal a-priori information on the harmfulness of the SCIUS. For instance, suppose that the lifetime of the SCE affected by the SCIUS is mostly volatile, therefore, it is likely that in general the SCIUS does not have a long term impact on the application.

Procedure: Mention the categories for which there are no empirical results. For the characteristics that have empirical results, consider a value typical when the percentage of SCIUS that have that value is greater than the percentage that would correspond to that category if the SCIUS would be equally divided among categories. For instance, if there are three categories for size, a value could be considered typical if it characterizes more than 33% of the SCIUS, and an atypical if it characterizes less than 33% of the SCIUS.

5.1.2 Causes and Consequences of the SCIUS

Given that the goal of the analyses proposed in this methodology is to assess the effect of having the SCIUS in a SCE, it is essential to describe the alleged impact of having that SCIUS as well as identifying which of those supposed effects have been verified. Nevertheless, the examination of the literature cannot be restricted to the consequences of having the source code issue, but also to its possible causes in order to distinguish correlation from causality connections in later analyses.

Figure 5-4 shows the steps required to complete this sub-phase.

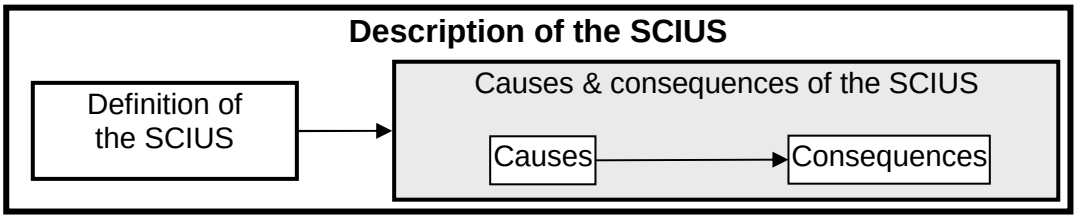


Figure 5-4. Description of the sub-phase "Causes and Consequences of the SCIUS" (in gray), from the phase "Description of the SCIUS".

5.1.2.1 Causes

Deliverable: A list of the possible causes for introducing the SCIUS. For each item on the list, there should be theoretical and empirical evidence to sustain or refute it. Aside from the evidence, each cause should have analysis of its relation to the SCIUS, as well as the likelihood of the cause provoking the SCIUS, and the likelihood of alternative causes provoking the SCIUS.

Rationale: The aim of this stage is to be able to find reasons for introducing the SCIUS that may overcome its claimed harmful effects. The causes of the SCIUS are relevant because they might be related to the consequences of the SCIUS. Besides, if instances of a SCIUS can be differentiated by their cause, their treatment can be prioritized. For instance, if performance is a key requirement in the applications and certain SCIUS are introduced to increase the performance and.

Procedure: Gather experimental and theoretical papers that discuss reasons for creating the SCIUS. For each cause, enumerate the arguments or empirical findings to sustain it or reject it. Discuss on the likelihood of having the SCIUS, given each cause based on the evidence

gathered.

5.1.2.2 Consequences

Deliverable: A list of the possible consequences of the SCIUS. For each item on the list, there should be theoretical and empirical evidence to sustain it or refute it. Aside from the evidence, each consequence should have brief analysis of its causality link to the SCIUS, as well as the likelihood of the SCIUS provoking the claimed effect, and the likelihood of alternative situations provoking the claimed effect.

Rationale: Classifying the works on the effects of the SCIUS into those that have built theoretical arguments, and those that have collected empirical evidence making a clear difference between facts and speculation. The analysis of the validity and reliability would state the level of corroboration of all of the claimed consequences of the SCIUS.

Procedure: Here one should collect the papers that mention the effects of the SCIUS. For each claimed effect, there should be a list of reasons, or empirical findings to sustain it or reject it. Finally, there should be a brief discussion to analyze the credibility of the claimed effect based on the strength of the evidence gathered.

5.2 Phase application

This section presents the results of applying the phase of the methodology explained in the previous section, i.e. the description of clones. The application of the phase is divided in two steps: the definition of clones, and the analysis of causes and consequences of clones. The definition and description of clones takes a significant portion of the section, because almost every paper that presents empirical results on cloning presents an ad-hoc classification of their results, and therefore there is a large variety of characteristics to summarize. The analysis of causes and consequences of cloning provides tables that summarizing the claims that have been made, as well as, the evidence found to support them.

To achieve each step, it is necessary to gather the studies that develop the topic to describe (i.e. definition, causes, or consequences of clones), and then cluster them according to their claims, procedures, or findings.

5.2.1 Definition of clones

A clone **relation** occurs when there is an identical code section except for a one-to-one correspondence of variables, constants, macro names, or member names [Baker '95].

Figure 5-5 highlights the near identical fragments. These fragments are called **cloned fragments** or **clone instances**. The parts that are similar between the fragments form a clone relation. There is a clone relation between the methods of Figure 5-5, that is between `m1(int, int, int)` and `m2(int, int, int)`. Clone relations form a correspondence between the variables: in the case of the example shown, the correspondence is a/x, b/y, and c/z. The clone formed by the clone relation of Figure 5-5 starts at the beginning of the parameters, and finishes at the keyword return. The clone does not finish at the end of the methods because there is no correspondence in the order of the return variables, and therefore a cannot be matched to y. Note that, although the layout and structure is identical, in the last statement the lack of correspondence makes the semantic to differ.

```
public int m1(int a, int b
, int c){
   if (a % 2 == 0){
      return a + b + c;
   }
   b *= 2;
   return a + b + c * 2;
}
```

```
public int m2(int x, int y
, int z){
   if (x % 2 == 0){
      return x + y + z;
   }
   z *= 2;
   return y + z + x * 2;
}
```

Figure 5-5. Clone relation example

Clones are a relation rather than an intrinsic characteristic of SCEs. A clone exists, if and only if, a similar fragment exists. In this thesis, we refer to **clone** as the common code in a clone relation. A clone is the longest sequential segment of code shared by at least two sections of code. Figure 5-6 shows the clone corresponding to the clone relation of figure 1. Note that the different parts are abstracted in the clone using a similar syntax to parameters using the special identifiers ***$1***, ***$2***, ***$3***. By replacing the corresponding parts, it is possible to obtain any of the fragments that form the clone relation.

```
                (int $1, int $2
, int $3){
   if ($1 % 2 == 0){
      return $1 + $2 + $3;
   }
   $2 *= $3;
      return
```

Figure 5-6. Clone corresponding to the clone relation in Figure 5-5

A **clone class** (**clone family** or **clone cluster**) is the set of code fragments that share the same clone. Figure 5-7 illustrates the clone family that forms the clone relation of Figure 5-5 by showing the method where the clone is located and the correspondence between the customizable parts (***$1***, ***$2***, ***$3***) of the clone and the actual methods where the fragments are located.

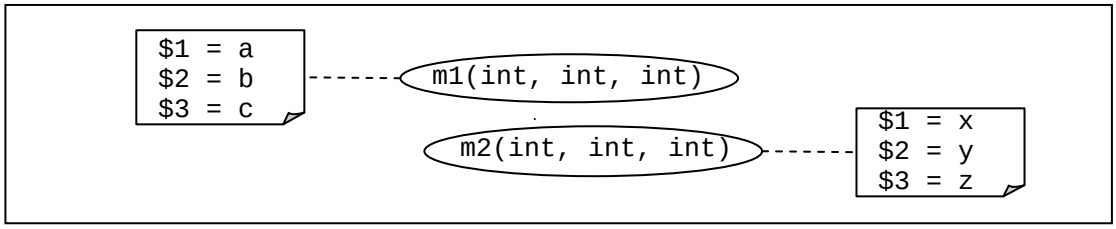


Figure 5-7. Clone class / cluster / family formed by the fragments highlighted in Figure 5-5

A fragment of a method is a sequence of tokens. There are two types of token: syntax tokens, and semantic tokens. **Syntax tokens** are those that cannot be instantiated, that is, keywords and operators. **Semantic tokens** are those that represent instances of SCEs; such as, literals, types, and method names.

Two **fragments are similar** if syntax tokens are identical, and if there is a correspondence in the type of their tokens. A correspondence in the type of tokens means that, for each syntax token in the first fragment, there is a syntax token in the second fragment; and for each semantic token in the first fragment, there is a semantic token in the second fragment. This means that our definition of similarity between fragments allows detecting clones type I[8] (or exact clones), and clones type II (or parameterized clones). Locating clones type III (or fragmented clones) depends on the length of the fragment that is defined to be minimal to consider the two

[8] For the definition of clone types, see Table 3-9.

fragments form a clone. If the minimal length of a clone fragment is short enough to detect fragments of code with some meaning, then this definition helps to find the fragments of clones type III. However, the clones would be detected as separate families.

Therefore, to define cloning inside methods it is not enough to define similarity with a fragment of another method. It is also necessary to state the minimum level of similarity required or the maximum level of dissimilarity allowed. In the case of our methodology, we have defined that **a clone occurs** whenever a method contains a fragment equal or greater than ***n*** tokens that is similar to a fragment in any other method in the application.

Establishing a minimum length for the similar fragments helps to avoid the analysis of meaningless clones. Note that if the length for similar fragments is unrestricted, most of the application would be considered cloned with fragments of one or two tokens.

However, our definition may require further refinement to reduce false positives[9].

According to experts [Kapser '07], a sequence of tokens is a clone if: the code is idiomatic, the code segments are very similar, there are lots of changes but identifiers remain very similar, it seems code copied and pasted, it seems generated code, the context of the code segment made it a clone, the fragments had the same structure. A sequence of tokens is a false positive if: it does not seem as copied and pasted code, there is no way to refactor it, the semantics of the code is meaningless, or if there many differences among fragments of the same family [Walenstein '04]. A fragment of tokens does not seem copied and pasted code when it is idiomatic. The semantic of the fragment is meaningless when it is too short. The semantic of different fragments changes when the types are different or when the majority of the fragment is made of literals [Walenstein '04].

Given that there is no consensus on what constitutes false positives [Walenstein '04; Kapser '07], our definition was adapted to include some considerations that experts use to eliminate false positives. **A method is cloned if**:

- its code is not generated

- it has a fragment of at least 30 tokens similar to another method in the application

[9] A **false positive** occurs when the test to detect phenomena falsely indicates that the phenomena occur.

- the number of tokens that differ from the fragment in the other method, with respect to the number of tokens of the fragment, is less than 45%
- the number of literals in the fragment is less than 45%
- the number of syntax tokens in the fragment is less than 45%
- and, the tokens in the fragment that refer to methods or types and that differ from the fragment in the other method is lower than 45% of the tokens in the fragment that refer to methods or types

This definition describes precisely the level of similarity required among tokens to consider that a fragment is cloned, because there is a minimum similarity required and a maximum number of differences tolerated among fragments. The minimum length of a sequence of similar tokens to be considered a clone is 30 tokens, and the tokens should be inside a method; therefore, the clones obtained are considered meaningful. Thirty tokens is a typical threshold for defining a clone when using token approaches [Kamiya '02; Kapser '03; Kapser '04; Kapser '05; Kim '05; Kapser '06a; Kapser '06b]. The maximum number of differences tolerated is described by two requirements: the different tokens cannot be more than the 45% of the tokens, the syntax and literal tokens cannot be over the 45% of the tokens, and the method or type tokens cannot differ in more than 45% of the method or type tokens. Besides, generated methods are excluded from the analysis.

The percentage of tokens that refer to methods or types that differ among the cloned fragments of the same family indicate a false positive because the semantics of the fragment in the method and of the fragment in the copy may differ significantly. For instance, if the most of the methods called or variable types referred are different it is likely that the supposedly cloned fragments have a different semantics and therefore they should not be considered clones.

The percentage of syntax and literal tokens also indicate false positives. If the majority of the cloned fragment is composed by syntax tokens, it might be that it is a common language structure for instance an *iteration* over a *collection*. If the majority of the cloned fragment is composed by literals, it might be that there is no relation between the supposedly cloned fragments but that it is a false positive due to using similar values for the literals.

These restrictions on the definition of a cloned fragment take into account several indicators that experts consider to identify a clone. From the restrictions suggested in the literature, we take

into account the ones that can be automatically validated, that is:

- code fragments are very similar: because at least 55% of the tokens should be identical
- the structure is the same: because all the syntax tokens should be identical
- types were changed: because the percentage of tokens that refer to types, methods, and literals, and that differ from other fragments in the family should be below 45%
- semantics of procedures: because the percentage of tokens that refer to types, methods, and literals, and that differ from other fragments in the family should be below 45%
- main portion of the code is literals: because the percentage of tokens that refer to types, methods, and literals, and that differ from other fragments in the family should be below 45%
- generated code is excluded from the detection of clones

5.2.1.1 Nature of clones

We have found that there are four ways to classify clones: according to characteristics of the cloned fragment (called classification C1 in section 3.3.1), according to characteristics of the relation between the cloned fragment and the method (called classification C2 in section 3.3.2), and according to characteristics of the cloned family (called classification C3 in section 3.3).

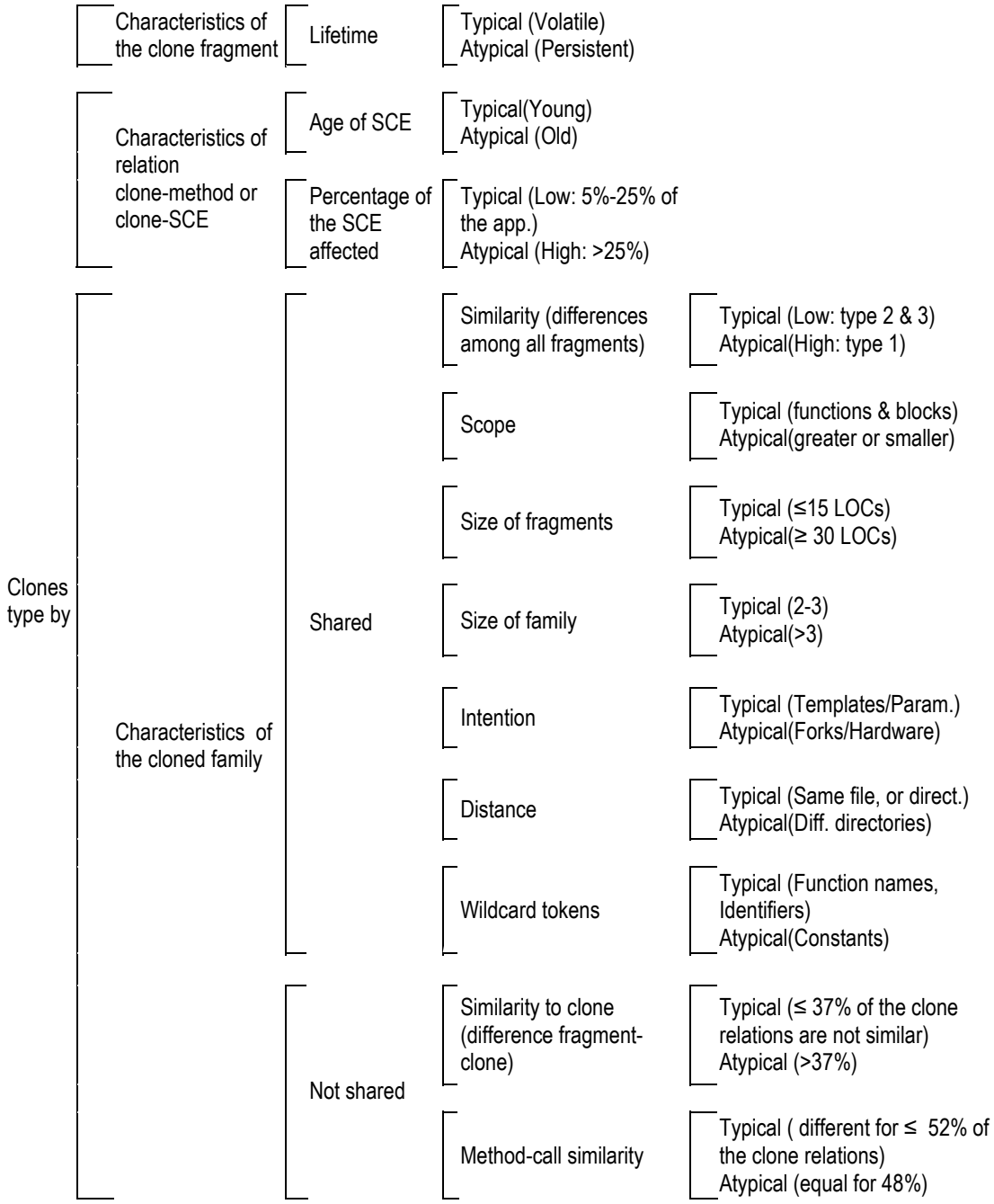


Figure 5-8. Nature of clones.

The characteristics of the clone family can be shared by all fragments in the family (presented as classification C3.1 in section 3.3.3) or not (presented as classification C3.2 section 3.3.4). A set

of characteristics that exemplify how clones have been classified, is shown for each type of classification, as shown in section 3.3.

The rest of the section explains for each type of classification, the characteristics used in empirical studies, the ways used to calculate the characteristics, and the empirical results obtained (summarized on Figure 5-8).

Previous empirical studies have shown that most clone families contain just a pair of cloned fragments, and have a high level of dissimilarity. The level of dissimilarity among fragments is evident in different ways: the cloned fragment is small, the gaps/wildcards are many or are large, the cloned fragments differ to the clone family, and clone fragments call different methods. Moreover, most of the clones are introduced in order to be modified. This is evidenced by several reasons: (1) most of the clones have a high level of dissimilarity, (2) most of the clones are introduced with the intention of being used as parameterized templates, (3) and clone families are small. Finally, it is likely that clones are introduced consciously because most of the clones were in close-by locations, and in small SCEs.

5.2.2 Causes and Consequences of cloning

This section presents a summary of the arguments to explain the existence of clones and their consequences. It shows that few of these arguments have been tested, and that for those tested, there is not yet enough evidence to support them. To understand this lack of evidence there is a discussion on the limitations of current evidence.

5.2.2.1 Causes

The literature describes several causes for cloning. We have divided such causes into two types: the one that has the developer to introduce the clone, the cause; and the reasons behind the developer decision. The summary of these causes is presented in Table 5-1. ‘A:’ indicates the papers presenting the arguments, while ‘E:’ indicates those presenting the evidence.

Table 5-1. Causes for cloning.

Cause - Justifications given
To use the cloned fragment as a template of new code. - Lower time in implementation A: [Johnson '94; Baker '95; Cordy '03; Giesecke '07]. E: [Kim '04] - Lower time in regression test (just testing new or changed code) A: [Johnson '94; Cordy '03]. E: [] - Trustworthy code A: [Johnson '94; Baker '95; Baxter '98; Cordy '03]. E: [Kim '04] - Code difficult to understand A: [Johnson '94; Giesecke '07]. E: [] - Productivity metrics are based on new lines of code instead of rewritten abstractions A: [Baker '95]. E: [] - Laziness. A: [Giesecke '07]. E: []
To overcome limitations of the available abstractions - Lack of abstraction mechanisms in the programming language issues (e.g. aspects) A: [Johnson '94; Giesecke '07]. E: [Kim '04; Basit '05b] - Inappropriate abstractions in the design A: [Johnson '93; Baxter '98; Ducasse '99]. E: []
To leave critical code untouched - High risk of changes A: [Cordy '03; Giesecke '07]. E: [Kapser '06a] - Restricted change on relations to keep the architecture neat and understandable A: [Cordy '03; Kapser '06a; Giesecke '07]. E: []
To improve performance - Relevant in embedded real time systems A: [Baker '95; Baxter '98; Giesecke '07]. E: []
Accidental cloning - Developers are not aware that they are implementing code already written A: [Lague '97; Giesecke '07]. E: [Al-Ekram '05] - Similar style A: [Baxter '98]. E: [] - Idioms A: [Baxter '98; Giesecke '07]. E: [Kim '04] - Similar declarations A: [Baxter '98]. E: [Kim '04] - Similar calculations A: [Baxter '98]. E: [Kim '04] - API usage A: []. E: [Al-Ekram '05; Li '06] - Crosscutting concerns A: []. E: [Kim '04] - Similar functionality A: [Baxter '98; Giesecke '07]. E: [Al-Ekram '05; Li '06] - Systems merged / Legacy code A: [Giesecke '07]. E: []

As mentioned before, there is a classification of clones by the intention of developers when cloning [Kapser '06a], i.e. a clone could be due to: forking, templating, customizing, and exact matches. Forking (i.e. leaving critical code untouched) is presented as a good type of cloning because it strives for stability. Some template clones (i.e. cloning by accident) are considered harmful because they require consistent changes along their lifetime. However, template clones are also considered good because they can improve the understandability of the code and they may be more costly to refactor than to maintain. Customizing (i.e. creating clones as templates) are considered good because the clone might be easier to understand than an abstraction. Finally, exact match clones (i.e. clones to overcome problems in the programming language) are considered good because of the type of validation that they introduce in the code and because they help to maintain the cohesion of functions.

This approach has two weaknesses. First, the rationale used for merging the causes for cloning with the types of clones is not clear. Second, the description of the types of clones is not detailed enough to unequivocally assign a type to a cloned fragment. The classification is subjective and the evaluation of the harmfulness for each type of clones has no empirical validation.

There is little evidence for ensuring that clones, in general, are created because of the reasons presented in the paragraph above or in Table 5-1. An approach taken to understand the rationale behind cloning decisions has been reflecting on the usage that industry gives to program comprehension and software maintenance tools [Cordy '03]. Cordy [Cordy '03] suggests that cloning in financial applications occurs because the functionality is similar. Programming based on existing code is encouraged to use trusted code and reduce the testing. Leaving critical functionality untouched is also promoted to reduce the risk on real-life financial operations, and refactoring is considered an investment with doubtful economic benefits. However, as the author says, these results may only apply to financial systems; even more, they may apply only to those financial systems analyzed. Another approach taken is logging the programming process, analyzing copy-and-paste operations extracted from the log, and confronting the researcher’s hypotheses with the developer’s reasoning for copying and pasting [Kim '04]. This approach has shown that developers clone to have semantic or structural templates, that programmers restructure cloned code once they are aware that they have pasted the same functionality twice, and that developers use cloning to implement crosscutting concerns [Kim

'05]. A third approach has been mining clones on large applications to confirm assumptions on clone creation [Al-Ekram '05; Li '06]. This approach has shown that clones are indeed used for API usage, and to implement similar functionality [Al-Ekram '05; Li '06].

5.2.2.2 Consequences

The literature describes as well several consequences of cloning. We have divided these consequences into two types: the characteristics that can be seen in the source code and the effects of those characteristics. The summary of cloning consequences is presented in Table 5-2.

Table 5-2. Consequences of cloning.

Observable consequence - Secondary effect A for Argument presented in [References]. E for evidence shown in [References]
Hidden relations. - Lower understandability A: [Johnson '94; Baker '95; Ducasse '99]. E: [] - Higher ripple effect, and therefore lower maintainability A: [Johnson '94; Baker '95; Mayrand '96; Ducasse '99; Monden '02]. E: [Lague '97; Kim '05; Geiger '06; Aversano '07; Krinke '07]
Larger codebase - Lower understandability, higher complexity A: [Johnson '94; Baker '95; Mayrand '96; Ducasse '99; Fowler '99; Giesecke '07]. E: []
Dead code - Lower maintainability A: [Johnson '94]. E: []
Logical bugs - Copied code is not adequately customized for the environment where it is pasted, e.g. incomplete renaming. A: [Johnson '94]. E: [Li '06] - Copied code has bugs A: [Johnson '93]. E: [Chou '01] - Copied code does not suit the requirements of the environment where it is pasted A: []. E: [Chou '01]
Degradation of design - There might be better abstractions for the application but they are not explored because cloning is a quick hack that works A: [Ducasse '99; Monden '02]. E: []
Critical, complex or working parts remain unchanged - Stability A: []. E: [Krinke '08] - Understandability A: [Kapser '06a]. E: []

Although there are plenty of speculation about the harmful effects of cloning, very few of these assumptions have been actually tested, and even less have been proven. So far, we know that:

Not all the changes to cloned fragments should be applied to the whole clone family [Lague '97; Krinke '07]

Maybe therefore co-changes could not be statistically correlated to cloning relations [Geiger '06]

Most of the clones disappear in less than 8 logical changes, and several of the ones that remain cannot be refactored [Kim '05], indicating that clones are either volatile or unavoidable.

Most of the inconsistent changes results in late propagations in a 24 hour interval [Aversano '07].

Inconsistent renaming is a typical error when cloning, and causes bugs [Li '06]

Clones are correlated with lack of knowledge of the application and contain bugs [Chou '01].

Cloned code seems more stable than code not cloned [Krinke '08].

5.3 Summary

This chapter describes the first phase of the methodology, which is the definition and description of the SCIUS. The first section explains the steps required to define the SCIUS, analyze its causes and effects, and describe of its common instances. The second section shows the results of applying the phase to analyze clones.

In spite of the important amount of work on clones, clones do not have standard definition. Therefore, we proposed a definition for clones inside methods that takes into account several of the recommendations from experts to recognize false positives. Although there are several studies that describe the clones analyzed (see section 3.3), trying to describe typical clone instances has not been attempted before. One of the challenges, and at the same time, weaknesses of integrating the results of the literature that describe clones is that there are different ways to measure the same characteristic. According to our integration of typical clone characteristics, clones are small snippets that are pasted in few, young SCEs in order to be customized to the new environment. Cloned fragments tend to be volatile with respect to their families. The differences between a cloned fragment and the cloned snippet that represents its family tend to be below 37% of the size of the cloned fragment (see Figure 5-8). Most of the

differences between a cloned fragment and the cloned snippet that represents its family are in names of functions and identifiers.

Furthermore, we have shown that even though there are many hypotheses to explain the existence and harmfulness of clones, there is empirical evidence for very few of them. In particular, there is no evidence that clones are introduced to: lower time in regression tests, increase understandability, increase productivity metrics, laziness, overcome problems in the design, follow architectural restrictions, improve real time performance, follow a particular style. In general, it seems that clones are introduced by accident rather than intentionally.

Moreover, there is no evidence for many of the effects attributed to clones. For instance:

- the relationship between hidden relations and the decrease of understandability
- the relationship between the increase in the code-base size and the reduction of understandability
- the relationship between dead code, due to clones, and lower maintainability due to that dead code
- the relationship between clones and design degradation
- the increase of understandability thanks to clones that eliminate over-complicated design relations on critical functionality areas.

Therefore, the long-term effect of clones on the ease to introduce changes (as templates) is still uncertain. Nevertheless, it is clear that cloned code is correlated with incorrect code , as some studies have shown that cloned code tends to have more bugs .

The next chapter presents the data collection phase; its steps, the challenges for collecting information, and the alternatives chosen for clones at method level.

Chapter 6. Applications to analyze

Describing the applications to analyze is the second phase of the methodology (see Figure 6-1) , which is also the second context phase.

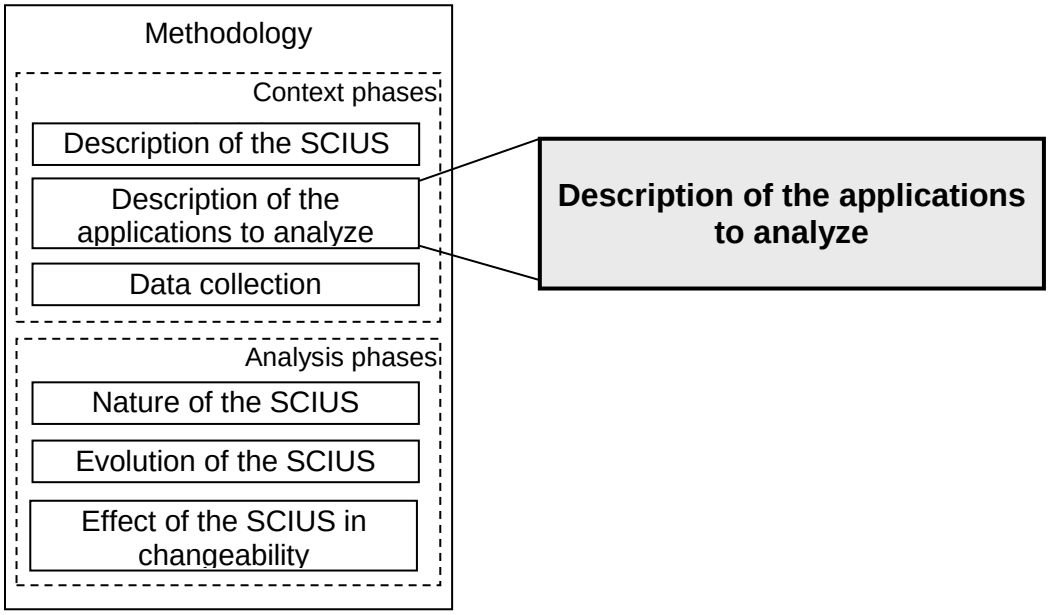


Figure 6-1. Second context phase of the methodology.

This phase does not answer directly any of the research questions, but it aims to establish the limitations of the findings. Identifying similarities and differences between the applications to analyze would describe the contexts in which the results are valid. If all the analyzed applications have the same value for a characteristic, it may mean that the results are valid only for the applications that have that value in that characteristic. For instance, suppose that the characteristic compared is programming language, and values for all applications to be analyzed are Java, the results of the applying the methodology are limited to applications developed on Java. On the contrary, if all the applications analyzed have different values for a characteristic, the results may be independent of the value of that characteristic.

The characteristics of the applications to analyze and the limitations they pose on the results should be considered when choosing the applications to analyze. Depending on the type of study designed, one may want to identify irrelevant characteristics or confirm results under certain characteristics.

Doing this phase first has not only the advantage of supporting the purpose of the experiments, but also of allowing the identification of factors that could help to discard accidental similarities in the results. Finally, the outcomes of this phase can be reused for all the experiments done over the same set of applications.

6.1 Phase description

This section presents the steps required to perform this phase. Before starting this phase, it is necessary to establish the settings of the methodology, i.e. the SCIUS, the SCE, and the applications to analyze.

Deliverable: The outcome of this phase is the set of similarities and differences among the case studies.

Rationale: Establishing the differences and similarities of the applications to analyze provides possible reasons that explain similarities or differences of the behavior of the SCIUS. This phase aims to analyze if characteristics of the applications affect the results or not. If the case studies differ in a characteristic, such characteristics may be irrelevant. If the case studies have a particular value for a characteristic, it is important to consider the effect of that characteristic on the SCIUS.

The output of this section can be used to foresee similarities and differences with respect to previous works.

Procedure: In addition to those characteristics defined when adapting the methodology to the SCIUS and the SCE, consider some basic system characteristics such as purpose, size, age, programming language, number of developers, and type of licensing of the application. Describe each case study according to those characteristics. Group the case studies according to the value of their characteristics. Summarize similarities and differences among the case studies for the characteristics suggested.

If possible, also describe the type of development process, as well as, major events on the

development of the application: change from closed source to open source, major restructurings, inclusion of new libraries or APIs, change of graphical interface, etc.

Adaptation: Decide, depending on the SCIUS and the SCEs chosen, which characteristics of the application may or may not affect the effect of the SCIUS on the changeability of SCEs. Formulate the expected behavior by relating an application characteristic with a changeability characteristic. For instance, the number of developers may be related indirectly with changeability. A high percentage of reuse may increase changeability because there is less code to maintain and less replication. The percentage of reuse depends on the chance that a developer knows if the functionality he needs is already in the program, where it is, and how to use it. However, the chance of knowing the application enough to reuse it reduces when the amount of developers increases.

6.2 Phase application

This section presents the results of applying the phase of the methodology explained in the previous section. The application of the phase is divided in two parts: the adaptation of the phase, and the application of the phase. In the adaptation part, we present the characteristics that could be linked to the effect of clones on the changeability of the methods that host them. In the application part, we present the results of comparing such characteristics in five case studies: Freecol[10], JEdit[11], Ganttproject[12], Columba[13] and JBoss[14].

6.2.1 Adaptation

Adapting this phase means deciding which characteristics of the application may or may not affect the effect of clones on the changeability of methods. The rest of this section explains why each of the suggested characteristics (purpose, size, age, programming language, number of

[10] CVS anonymous access: anonymous@freecol.cvs.sourceforge.net:/cvsroot/freecol

[11] There is no CVS access available since the restructuring of the application, and its migration to SVN. The developers gave us a copy of their CVS repository.

[12] CVS anonymous access: anonymous@ganttproject.cvs.sourceforge.net:/cvsroot/ganttproject

[13] CVS anonymous access: anonymous@columba.cvs.sourceforge.net:/cvsroot/columba

[14] CVS anonymous access: anonymous@anoncvs.forge.jboss.com:/cvsroot/jboss

developers, and type of licensing of the application) should be taken into account to discard accidental similarities in the results.

6.2.1.1 Purpose

The purpose of the applications could indicate if certain domains are more prone to cloning than others. If the chosen applications have the same purpose/domain, and they have similar cloning behavior, it is likely that the results are not valid for applications of another domain. If applications have the same purpose/domain and different cloning behavior, it means that it is likely that the domain is not related with the cloning behavior. If the chosen applications come from different domains and they have a similar cloning behavior, it would mean that the cloning behavior is independent of those domains. If applications have a different purpose and different cloning behavior, it is impossible to conclude anything relating the domain and cloning. It is believed that applications in certain domains may be more affected by cloning than others. For instance, operating systems may have more clones than other applications because drivers are developed by copying a similar driver that already works, and customizing the copy according to its context. Another example is financial applications: they are developed as updates or enhancements of similar functionalities. This happens because products in the same financial institution do not vary much [Cordy '03]. In addition, copying and adapting is encouraged in financial applications because restructuring is a high-risk operation, and existing code is highly trusted due to careful and extensive tests [Cordy '03].

6.2.1.2 Size

The size of the application could show a relation between the size of the application and its amount of cloning, i.e. whether the application is unnecessarily large because of the redundant code that clones inject. Very large applications may have similar functionality implemented several times because developers are unaware of its existence. However, if small applications have a similar percentage of cloning to large applications, it might be that cloning occurs in that percentage of cases regardless of the size of the application.

6.2.1.3 Age

The age of the application could also play an important role because developers may create clones with unfamiliar code, in this case, very old code. Moreover, the age of the applications

could show if harmfulness of clones is related to the amount of time that they have being part of the application, and if so, whether the relation is direct or inverse. Moreover, the age of the application affects the amount of changeability data it offers. It is possible to consider applications of different ages; nevertheless, they should have a long enough history to find several cases of cloning and to be able to analyze changes in those clones.

6.2.1.4 Programming language

The programming language could have a very big effect on the analysis of clones. It is likely that some languages have higher quantities of particular types of clones related with abstractions that are not available in the language, e.g., abstractions to implement crosscutting concerns. If the effect of clones is related to the type of clones, the fact that the distribution of types of clones varies from one language to other may change completely the overall effect of clones.

6.2.1.5 Number of developers

The number of developers can also affect the results. It is unlikely that in small groups of developers, developers create clones by accident, which means that the clones are more likely to be consciously created. It is also unlikely that in a small group, developers are unaware of existing cloning. Therefore, it is unlikely that developers change a fragment of code without changing its clones in the same way. Hence, the number of developers in an application may have a strong influence of the changeability of cloned methods.

6.2.2 Application: comparison of the applications to analyze

In order to apply the methodology, we developed an application to gather the data, process it to provide intermediate results, and perform a number of automatic analyses. The tool is a proof of concept for the methodology that relies on third party applications for parts of the processing. Our application was implemented to analyze Java code on CVS repositories. CVS was chosen among the variety of SCM applications because it is old, and popular; hence, it is likely that many applications that have a long history use CVS. Java was chosen in order to facilitate the comparison of our results with other empirical studies of clones given that many of them analyze Java code [Kamiya '02; Monden '02; Kim '04; Kim '05; Kapser '06a; Li '06; Aversano '07; Baker '07; Krinke '07]. However, this means that our results are limited to applications written in Java.

Table 6-1. Characteristics of the applications to be analyzed.

		Freecol	JEdit	Ganttproject	Columba	JBoss module
	Purpose	Game	Text editor	Planning tool	Mail client	J2EE app. server
Size	**KLOC**	53	98	44	92	2
	#methods[@]	3252	5277	3558	7441	179
Age	**#months**	35	58	43	52	30
	#commits	1087	1381	2701	3108	3346
	#committers	14	13	20	16	86

[@] Number of methods on the last version analyzed of the application.

As mentioned above, the tool implemented poses two restrictions on the applications to be analyzed: being implemented in Java, and being stored in CVS repositories available to anonymous users. We decided to choose applications from different domains because *we suppose that the nature of cloning should be the same regardless of the type of functionality implemented*. We selected the projects with varying age, size, and number of developers to obtain a wider view of the cloning phenomenon. FreeCol is a game in which players have to conquer and colonize new worlds. JEdit is a text editor for programmers that can be configured as an IDE through its plug-in architecture. GanttProject is a project scheduling application with facilities for doing Gantt charts, resource management, calendars, etc. Columba is an e-mail client with graphical interface and internationalization support. JBoss is a J2EE based application server, from which we analyzed the JBoss module. The description of these applications in terms of purpose, size, age, and number of developers is summarized in Table 6-1.

Describing the applications to analyze involves finding some characteristics that may affect their changeability, and deciding in which aspects they are similar, and in which aspects they differ. Given the variability of numeric values, they may not be good indicators of similarity or dissimilarity. To group the case studies by the values of a characteristic, we propose to categorize the values analyzed depending on the range of values across the applications analyzed. The most common value would be the medium category, those above it would be the large category, and those below it would be the small category. The result of this categorization of the case studies is shown in Table 6-2.

Table 6-2. Characteristics of the applications to be analyzed.

	Freecol	JEdit	Ganttproject	Columba	JBoss module
Size (KLOC)	Medium	Large	Medium	Large	Small
Size (#methods)	Medium	Large	Medium	Large	Small
Age(#months)	Small	Medium	Medium	Medium	Small
Age(#commits)	Small	Medium	Medium	Large	Large
No. developers	Medium	Medium	Medium	Medium	Large

In terms of size, the applications fall in three categories. JBoss which is the smallest, followed by Freecol and Ganttproject that are in the same level, and above them JEdit and Columba. However, note in Table 6-1, that the ratio between the lines of code and the number of methods differs; so, it is likely that the modularity of these applications differ. For instance, it seems that JBoss has less lines of code per method than the other applications. The contrary situation would apply to JEdit, Ganttproject, Freecol, and Columba, which seem to have a similar number of lines of code per method.

In terms of age, the factors analyzed differ. Although Columba is in the medium category regarding number of months analyzed, regarding the number of commits it lays in the large category regarding number of commits. Similarly with JBoss that belongs to the small category in terms of months analyzed, but to the large category in terms of commits. This indicates that JBoss has the highest activity rate of the applications, while Columba has the second higher activity rate. The number of logical changes alone can also indicate how active the project is. However, this is only valid if all projects make logical changes in the same way, i.e. when one or more features are added, changed, or deleted. It might be the case that developers use the SCM system as a backup tool instead of a way to synchronize their contributions with other developers.

Notice that, in recent years, many OSS have migrated their SCM system from CVS to SVN, among other reasons because SVN is more modern and handles directly concepts like logical changes that are not native in CVS. Therefore, in some cases, the interval analyzed does not reflect the current age of the application but rather on their lifetime stored in CVS.

The only issue regarding old applications is that the forces that drive their evolution have changed significantly and the insights found for the initial changes might not be applicable for the latest changes. Nevertheless, analyzing applications with long histories provides more information to analyze, and more background to incorporate to the analysis.

Finally, regarding the number of committers, one could think that Freecol, JEdit, Ganttproject, and Columba have a similar behavior in terms of amount of developers. However, it is important to keep in mind that the number of committers may not be the same as the number of contributors. In some OSS very few contributors have direct access to the repository in order to have quality control on the contributions made by the community. In any case, a large amount of developers, such as the one found in JBoss, may indicate that there are more clones because there is less awareness of other developer's code.

6.3 Summary

This chapter presents the first step of the methodology, which is describing the similarities and differences among the case studies to have at hand arguments to evaluate the limitations of the results.

This chapter justifies the characteristics used to compare the applications to analyze based on their usefulness to eliminate or confirm hypotheses between the effects of clones and the nature of the application. There are three reasons to justify the characteristics chosen: the types of cloning in an application, the awareness of developers about the clones, and the management of changes and clones depend on characteristics of the application. The types of cloning that are likely to appear in an application may depend on its purpose/domain, and its programming language. The likelihood of cloning unwittingly may depend on the size, age, and number of developers of the application. Finally, the type of licensing may play a role in the policies or lack of policies on handling clones.

The applications selected to apply the methodology were chosen by the type of SCM repository (CVS), the availability of their SCM repository (Sourceforge), and the programming language (Java). JBoss has several characteristics in which they differ from the rest of applications to be analyzed. According to the heuristic used to compare the characteristics, the applications to analyze differ on the number of LOCs, number of methods, length of the history analyzed (months), activity rate (commit transactions over time interval), and the size of the development team. The only characteristic that most of them share is a similar number of developers, with the exception of JBoss. We have concluded that Columba and JBoss have a higher activity rate than the rest of the applications, and that the modularity in these applications is better than in the rest. Apart from that, JEdit and Columba have a comparable size, different from the size of Freecol

and Ganttproject, which is comparable between them.

Once the applications to analyze are described, we can proceed to analyze the current knowledge on the SCIUS. The next chapter explains the second step of the methodology: the definition and description of the SCIUS; in this case, the definition and description of clones.

Chapter 7. Data collection

This chapter explains the third phase of the methodology, which is also the last of the context phases (see Figure 7-1). This phase is called data collection. The phase aims to gather historical data on the status of the SCE, and on the changes of the SCE. The data gathered should be enough to compare the changeability of SCEs with and without the SCIUS, and assessing if any change on the changeability could be due to external factors like characteristics of the SCEs or of the SCIUS.

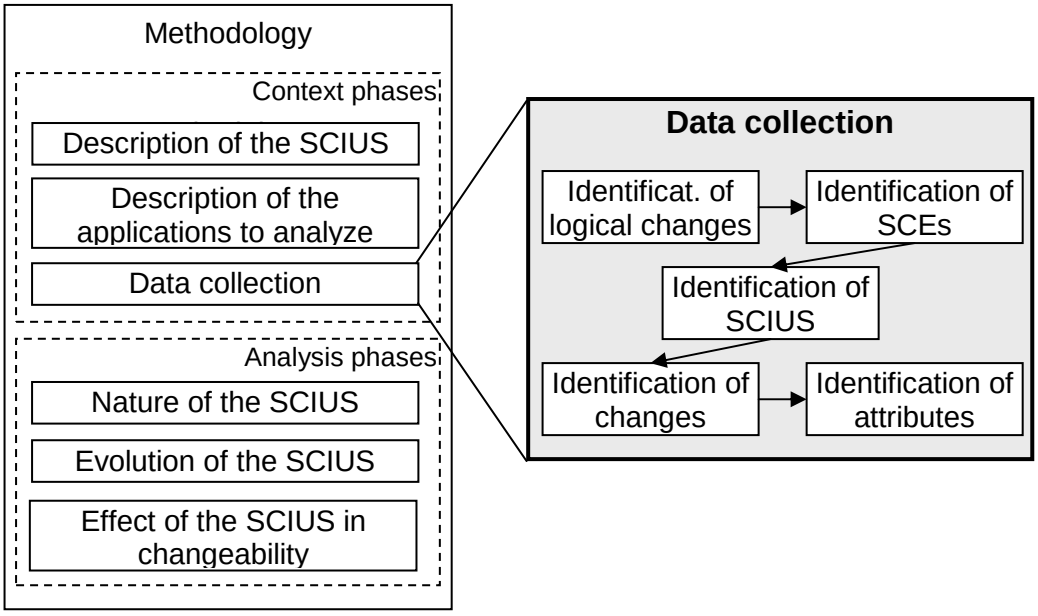


Figure 7-1. Description of the phase "Data collection".

7.1 Phase description

In this section, we explain the steps required to follow to ensure that the data collected is enough to apply the methodology, and that it accurately reflects the hypothesis.

The data collection phase is composed by five sub-phases: identification of logical changes, identification of SCEs, identification of SCIUS, identification of changes, and identification of

attributes (see Figure 7-1). That means collecting along the lifetime of an application information about: the SCEs that compose the application, the SCIUS that those SCEs have, the changes that affect the SCEs and the SCIUS, the characteristics of the SCEs, and the characteristics of the SCIUS. Each sub-section below explains how to complete each step.

7.1.1 Identification of logical changes

The details on how the literature proposes to gather the history of an application are explained on section 2.3.1. Collecting the historic information is done in five steps, shown in Figure 7-2.

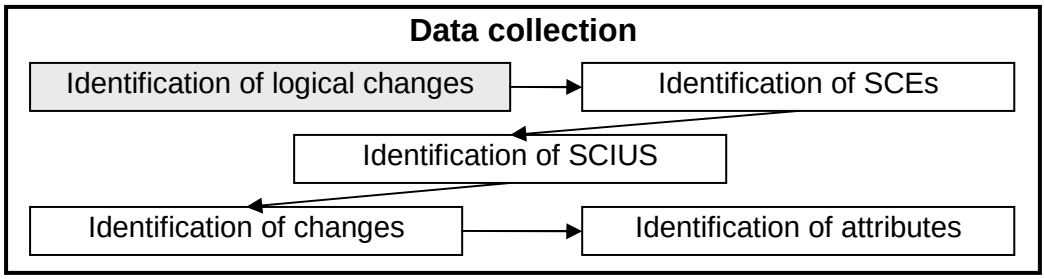


Figure 7-2. Location of the sub-phase "Identification of logical changes" in the phase "Data collection".

Deliverable: A list of logical changes per case study (see the definition on section 4.2). The following characteristics of logical changes should be captured:

- number of logical changes,

- distance in time to the next logical change,

- percentage of logical changes per author,

- and percentage of SCEs changed per logical change.

Rationale: Analyzing logical changes instead of physical changes exposes the complexity of updating an application, because it indicates the complexity of changes. Logical changes indicate the relevant points in time in which the application changed. In order to have reliable data, the SCEs that compose the application, and the changes that have occurred to those SCEs should be obtained after each logical change.

Procedure: SCM applications that do not store logical changes often have enough information to reconstruct them from the information about physical changes. In order to detect logical changes one should classify the changes in three ways: time-wise, author-wise, and intention-

wise. Those changes that are stored in a close time range, under the same author and intention belong to the same logical change. First, changes should be classified by author and intention. The time-closeness can be defined by fixed or sliding time windows as explained in section 2.3.1.

7.1.2 Identification of SCEs

Once logical changes are identified, it is necessary to iterate for each one of them to reconstruct the history of the application. The first step to process a logical change is to identify the SCEs that compose the application.

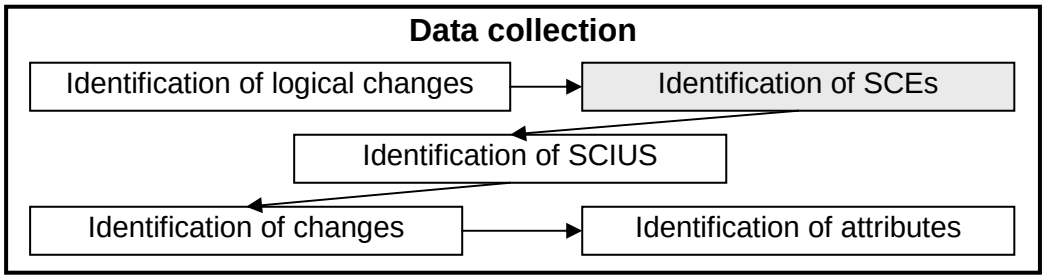


Figure 7-3. Location of the sub-phase "Identification of SCEs" in the phase "Data collection".

Deliverable: The structure of SCEs that compose an application after each commit transaction. Before starting to apply the methodology it is necessary to define which SCEs are going to be analyzed: this step permits us to establish the minimum level of granularity of SCEs to gather. That means, the possible units of analysis are at that level of granularity and upwards, i.e. the SCEs that are containers of the SCEs that are going to be analyzed.

Rationale: The analysis of changes across SCEs at different levels of granularity could indicate to what extent the structure of the application supports the changes that the application undergoes. If the changes affect SCEs that are not enclosed by an abstraction, it means that the structure is not hiding the details of changes, and therefore that the structure is supporting the changeability of the application. Contrarily, if changes are local, then it is likely that the structure of SCEs supports the changeability of the application.

Procedure: For each logical change, reconstruct the SCEs and their upper structure by analyzing the files stored in the SCM repository and using the syntax of the programming language.

Adaptation: This step requires being adapted depending on the granularity of the SCEs to

analyze.

7.1.2.1 Tracking of SCEs across snapshots

Deliverable: For each snapshot (i.e. logical change or commit transaction), a set of SCEs that were renamed.

Rationale: This step is necessary when the history of a SCE is tracked using its identifier i.e. their fully qualified name. It is necessary to merge the history of those SCE that were apparently deleted with those that were apparently created but that in fact are the same SCE with different identifiers.

Procedure: SCEs can be automatically tracked locating their identifier over time. The idea is to use this naïve approach to track an initial version of the history of SCEs, and afterwards, refine it by locating the SCEs whose identifier changed. Therefore, it is necessary to validate for each snapshot which SCEs were indeed deleted, and which of them just changed their identifier. There are two alternative means of finding out if two SCEs (one of the previous snapshots, and one of the current snapshot) are the same SCE but with a different identifier: using parts of its identifier, or using the SCEs that compose it.

Adaptation: This step should be adapted by specifying which parts of the identifier, or components of the SCE, to use for the comparison, and how to compare them.

7.1.3 Identification of SCIUS

Once the SCEs that compose the application are identified, it is necessary to identify which of them are affected by the SCIUS. This is the second step to process a logical change.

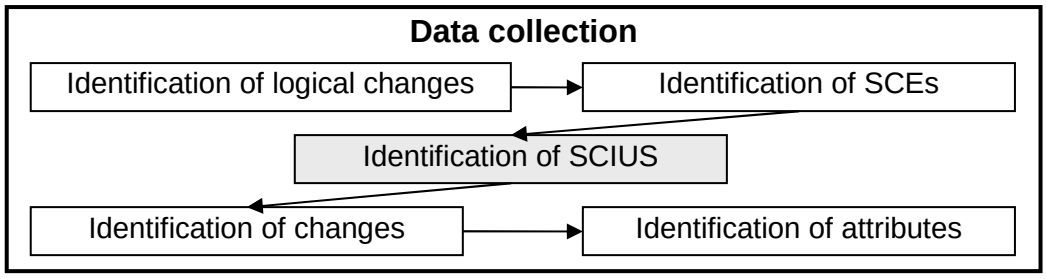


Figure 7-4. Location of the sub-phase "Identification of SCIUS" in the phase "Data collection".

Deliverable: List of algorithms to locate instances of the SCIUS, as well as the tools that implement them. Relevance of each tool with respect to the definition chosen for the SCIUS. A

list of the restrictions of each tool (e.g., programming languages that it can handle). A comparison of precision and recall of those tools (if there are studies available in the literature).

Rationale: The quality of the results depends on the quality of the data analyzed. In order to have an accurate and complete set of exemplars of the SCIUS it is important to be sure that the tool or algorithm used to detect them is the one best suited for the definition of the SCIUS (high precision). It is also important to find most of the SCIUS (high recall) to be able to compare the SCEs with the SCIUS against those without them.

Procedure: Gather papers whose goal is to identify the SCIUS. Group the papers by algorithmic approach. Compare the advantages and disadvantages of each algorithmic approach. Enumerate the tools that implement each algorithmic approach. Gather papers whose goal is to compare tools that locate SCIUS. For each tool, discuss the appropriateness of the tool to locate SCIUS as defined for the methodology, and report its precision, recall, restrictions, and advantages.

7.1.3.1 Tracking of SCIUS across snapshots

Given that SCIUS do not necessary belong to SCEs, it is necessary to be able to track them exactly in which area of the SCE where they are located. Tracking SCIUS only by their location can result in inaccurate data given that any SCE may have several SCIUS. Although for several analyses this might not be a problem, for detailed analyses it might lead to incorrect data. For instance, when deciding if the SCIUS was changed, or in analyses that take into account the characteristics of the SCIUS because it is necessary to distinguish which SCIUS is being analyzed. Therefore, it is important to store any other information that is unique from the SCIUS to identify it. For instance, which method and class are related by each instance of feature envy, or which is the clone that a cloned fragment parameterizes.

7.1.4 Identification of changes

Once the SCEs that compose the application and their SCIUS are identified, it is necessary to identify which of those SCE have changed with respect to the previous logical change/snapshot. This is the third step in processing a logical change.

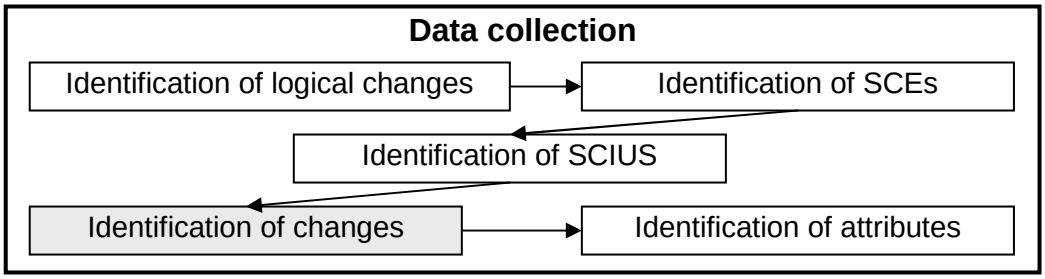


Figure 7-5. Location of the sub-phase "Identification of changes" in the phase "Data collection".

Deliverable: The history of changes of the application. That is, the whole set of logical changes that an application has undergone, and for each one of them: the SCEs that compose the application, the SCEs that changed, and the SCIUS that changed.

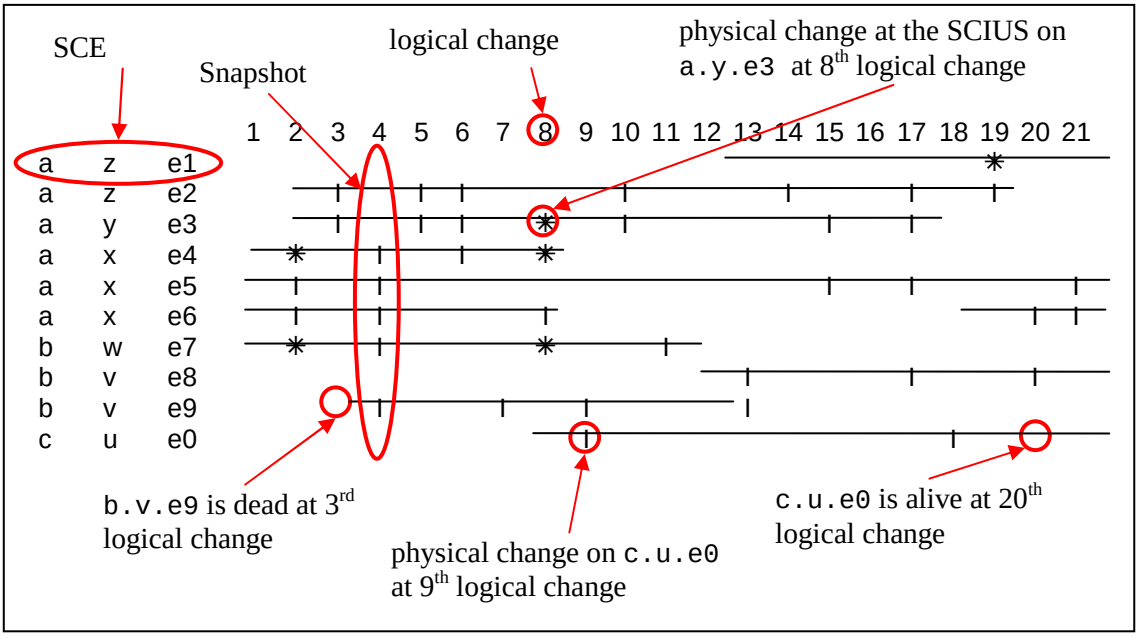


Figure 7-6. SCE history for a fictitious application.

Figure 7-6 illustrates the history of changes in the application. The rows represent SCEs. The columns represent logical changes or commit transactions. Logical changes are also called snapshots of the history of the application. Each cell, i.e. the intersection of a row and a column, indicates what happen to that SCE in that logical change. If the SCE was not part of the application, the cell is empty. In this case, we say that the SCE is dead. If the SCE is part of the application but does not change in that logical change, the cell has a hyphen. In this case, we say that the SCE is alive. If the SCE is part of the application and changes in that logical change, there is a cross in the cell. If the SCE is part of the application, and the SCE changes in that

logical change, and the change occurs in the area of the SCE that has the SCIUS, there is an asterisk in the cell. Notice in the figure that although there are several SCEs that come into existence after the first logical change, they do not appear as changed: this happens because the creation of a SCE is not considered a change.

Rationale: Changes are traced at the level of SCE to assess the extent of changeability alteration on the SCEs in periods with SCIUS. Changes are traced at the level of SCIUS to measure the percentage of changes attributable directly to the SCIUS.

Procedure: Identify the logical changes that the application has undergone. For each logical change, identify the SCEs that belong to the application. For each SCE that is part of the application in that logical change, find if there is any physical change to that SCE (i.e., if the SCE on the previous snapshot differs from the SCE on the snapshot corresponding to the current logical change. For each SCE that has changed, identify if the components of the SCE that were modified are among the components of the SCE that contain the SCIUS.

Adaptation: This step requires being adapted to the SCEs and the SCIUS to analyze. The granularity of changes stored on the SCM depends on the type of SCM system of the application analyzed. Most SCM do not store syntactic information, so they would not been able to identify the SCE or the SCI changed. In fact, it is likely that the only information about changes available are the lines changed from one version to another version of the same file. Therefore, it would be necessary to convert from lines of code to SCEs or SCIUS modified.

7.1.5 Identification of attributes

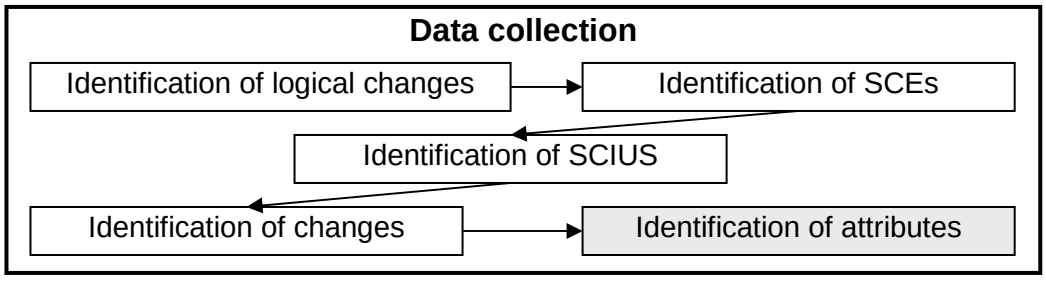


Figure 7-7. Location of the sub-phase "Identification of attributes" in the phase "Data collection".

This sub-phase consists of storing the characteristics of SCEs and of SCIUS that may affect the changeability of the SCEs. This is the last step to process any logical change, but also the last sub-phase of the 'Data collection' phase, as Figure 7-7 shows.

Deliverable: The history of the attributes of the SCEs, and of the SCIUS.

Rationale: The attributes of SCEs could help to identify characteristics that may explain the changeability of the SCEs regardless of the SCIUS. The attributes of SCEs and of SCIUS could help to identify characteristics that increase or decrease the effect of the SCIUS in the SCE.

The attributes analyzed should be calculated automatically, and given that they have to be computed for all logical changes, their calculation should have a good time performance.

Procedure: Select attributes of the SCIUS that may be related to the changeability of the SCE, it is likely that these attributes are related to the consequences of the SCIUS that were analyzed on section 5.1.2. Select attributes of the SCE that might be linked to the changeability of the SCE. For the SCEs add a boolean attribute to indicate if the SCE had the SCIUS at that snapshot or not.

Remove from the lists of candidates those that cannot be calculated automatically, those whose calculation is not deterministic, and those that cannot be assigned to a single SCE or to a single SCIUS. For each of the remaining attributes determine the type of attribute: nominal (no order), ordinal (ordered with uneven intervals between scales), or interval (ordered with even intervals between scales); and the type of data: text, boolean, or numeric (if it is numeric state if it is natural, or real). For each attribute, indicate the tools or algorithms used to obtain it.

7.2 Phase application

The data collection algorithm finds the logical changes, and then, for each logical change, it stores in a database the information associated to the resulting snapshot. The information stored per snapshot includes files, classes, and methods, methods that changed and in which lines, methods that have clones, information on the clones, information on the methods, etc.

The algorithm of the tool that gathers the data is presented below (see Figure 7-8). All the data per application analyzed is stored in a database. The data collected is stored in a database because it permits expressing queries easily, which is an advantage for the analysis.

```
Collection of logical changes
For each logical change
      Extract snapshot from CVS
      Find methods
      Find clones
      Find clones in methods
      Find changes in methods
Find changes in clones
Compute characteristics of methods
Compute characteristics of clones
```

Figure 7-8. Data collection algorithm

This section explains the data collection algorithm for the basic historic information, i.e. methods, clones, and changes, then for characteristics of clones and methods. As mentioned previously, we decided to analyze applications stored in CVS repositories; this means that it is necessary to identify logical changes, the methods modified by each logical change, and to perform origin analysis. The data collection was implemented in an algorithm that takes the information of logical changes to reconstruct the snapshots of the application. For each snapshot, the algorithm detects the methods, and their attributes, clone relations, and clone families using the output of third party tools. The methods and the SCEs above them are identified using a lexical analysis tool. The clone relations and clone families are obtained from the clone detection tool used. However, the data collection algorithm is responsible for relating cloned fragments and methods, and for tracking methods, cloned relations, and cloned families. The identification of the lines of code cloned and the methods and clones changed are also implemented in the data collection algorithm. A third party static analysis tool obtains the characteristics of the methods, while the data collection algorithm detects the characteristics of the clones. This section explains how all this data is obtained and stored so the analyses phases can be executed.

7.2.1 Identification of logical changes

CVS is a SCM application that does not store logical changes. However, CVS provides a time stamp for each physical change on each file. Each time stamp contains the exact date and time in which the change was stored in the repository, as well as the identifier of the person that stored the change, and the message submitted by that person with the rationale for the change.

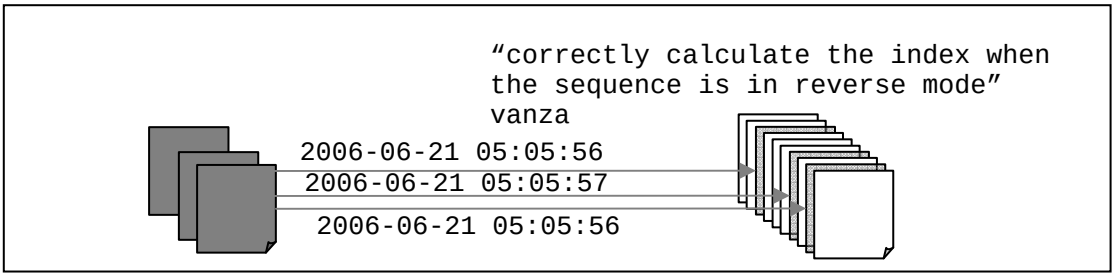


Figure 7-9. A commit transaction as stored in CVS. Several files have the same author, message, and a close time stamp.

Figure 7-9 has a graphic description on how commit transactions are stored in CVS and how they can be identified in CVS using the similarities among the log of files. This figure shows that three files were modified by the developer identified with the id “vanza”; the changes in those files had the same message (in the top-right corner of the figure), and the time stamps of the changes were separated by one second.

That information is extracted from the CVS log file. The screenshot of the CVS-Log file of the file `SegmentCharSequence.java` that includes one of the individual changes that compose the commit transaction of Figure 7-9 is shown in Figure 7-10. The lines 15 through 17 in Figure 7-10 correspond to the change in the file `SegmentCharSequence.java` that is part of the commit transaction depicted in Figure 7-9 .

In order to obtain the logical changes, we obtain first the log of all files that have been stored in CVS as part of the application. The log is a concatenation of logs of each file; a log of one file is shown in Figure 7-10. This log is obtained by calling the log command on the root of the application, i.e.

```
cvs log .
```

Log files are usually very long. Therefore, once the whole file is loaded into a sector of the memory, it is parsed automatically to group the timestamps author-wise and message-wise. This grouping is done by creating a list of unique pairs author – message found in the log. For each pair is stored a list of physical changes. For each physical change its timestamp, file, revision of the file, and if the file was deleted or not is stored. Once all timestamps are in groups that share the same author and message, these groups are subdivided in more groups time-wise to obtain the commit transactions.

```
RCS                                                                  file:
/cvsroot/jedit/jEdit/org/gjt/sp/util/SegmentCharSequence.java,v
Working file: org/gjt/sp/util/SegmentCharSequence.java
head: 1.2
branch:
locks: strict
access list:
symbolic names:
   after_bsh-2-0b4: 1.2
   before_bsh-2-0b4: 1.2
   jedit-4-3-pre5: 1.2
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
----------------------------
revision 1.2
date: 2006/06/21 05:05:56; author: vanza; state: Exp; lines: +1 -
1
correctly calculate the index when the sequence is in reverse
mode
----------------------------
revision 1.1
date: 2006/06/18 18:51:40; author: vanza; state: Exp;
remove all dependencies on gnu.regexp
```

Figure 7-10 . CVS log of the file SegmentCharSquence.java. Each change to the file is identified with a number and several characteristics of the change like date, author, and rationale for the change

Grouping the physical changes time-wise is done using a sliding time window (see section 2.3.1) of three minutes [Zimmermann '04; Kim '05]. This means that for a physical change to be added to the commit transaction, it is at most three minutes later than the most recent timestamp in the commit transaction. The final set of groups of physical changes organized from the earliest to the latest is the set of commit transactions or logical changes. Each commit transaction is composed of a set of physical changes saved by the same author, with the same message, and whose changes were stored in at most three minutes among them, i.e. the set of physical changes stored in the same transaction. Given the information stored during the process of identification of commit transactions, each commit transaction knows its exact start and finish time, its author, the rationale of the change, and which files where deleted or modified.

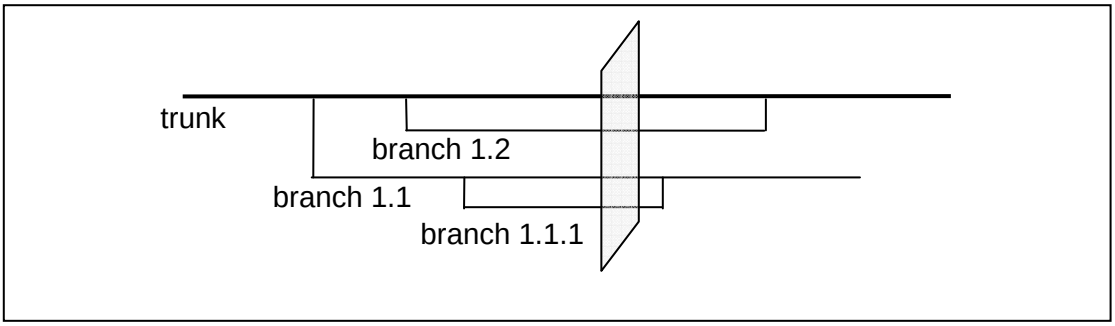


Figure 7-11. CVS permits storing different versions of the application at the same moment in time. The main history is in the trunk, while support / testing changes are stored in the branches.

CVS changes are stored in history lines. However, there are changes that can deviate from the main application, which are stored in parallel history lines (as shown in Figure 7-11). For instance, changes required to test the usefulness of a new library instead of integrating it directly to the main code base. These parallel lines of history are called **`branches`** in CVS. Note that having branches implies that a file can have several versions at a moment in time (see in Figure 7-11 the gray plane that cuts the history). At those, and any of the subsequently moments it is uncertain which version of the file is the correct one. Taking into account that changes in branches may introduce errors in the detection of methods or clones, and of changes, the algorithm ignores branches, processing only changes in the main trunk of the history. The logical changes eliminated from the analysis for being in branches instead of in the main trunk are summarized in Table 7-1.

Table 7-1. Types of clones by the distance the cloned fragments in the family

Application	Logical changes in the period analyzed	Logical changes analyzed
Freecol	1090	1087
JEdit	1382	1381
Ganttproject	2943	2701
Columba	3136	3108
JBoss module	3967	3346

The information about logical changes is stored in three tables on the database: the *developer* table, the *commitTransaction* table, and the *commitAuthor* table, which are shown in Figure 7-12.

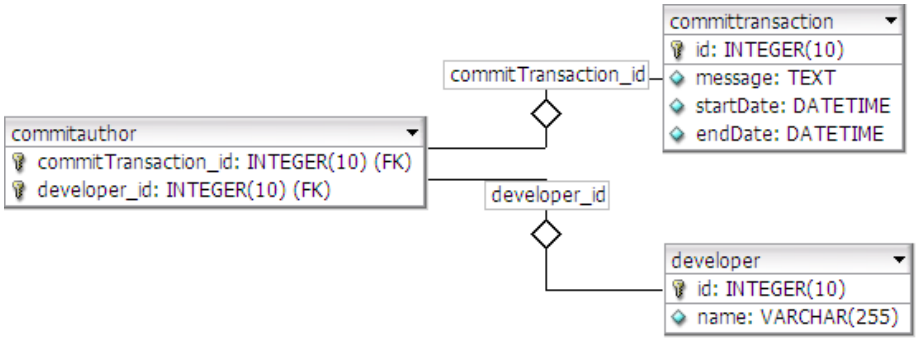


Figure 7-12. Tables that store the information of logical changes in the database.

Once the logical changes are identified, it is possible to download the snapshots of the application in order to gather the history of the application. Downloading snapshots in CVS is done with the `checkout` command. The data collection algorithm defines a directory in which the application will be updated to reconstruct the snapshot corresponding to the changes after each logical change. Given that, among the information stored in memory while identifying the logical changes, is the set of files affected by the logical change, the algorithm can overwrite only those files that changed by downloading their status by the time the logical change finishes. An example of the checkout commit executed with this information is shown below:

```
cvs checkout      -A
                  -D 2006/06/21 05:05:57 GMT
                  org/gjt/sp/util/SegmentCharSequence.java
```

This command would replace the local version of the file `SegmentCharSequence.java` by the version in the CVS repository on the 21st of June of 2001 at 5:05:57. Once the snapshots of the application after each logical change are obtained, it is possible to identify the methods, classes, files, clones, and their characteristics at any moment of the history of the application.

Table 7-2. Summary of logical changes in the analyzed applications

Application	No. of logical changes	Avg. time separation between logical changes	Avg. percentage of methods changed per logical change	Percentage of logical changes per author (top 3 authors)
Freecol	1087	23 hours	0.1 %	44%, 24%, 15%
JEdit	1381	1 day and 5 hours	0.09 %	82%, 6%, 4%
Ganttproject	2701	12 hours	0.1 %	48%, 37%, 6%
Columba	3108	6 hours	0.04 %	48%, 19%, 18%
JBoss mod.	3346	10 hours	0.05 %	13%, 12%, 11%

The results of the extraction of logical changes are shown in Table 7-2.

7.2.2 Identification of methods

Here we present the approach used for identifying methods in the application we developed. The approaches for method identification discussed in the literature, their advantages and limitations are summarized on section 2.3.1.1.

CTAGS[15] is an open source application that uses lexical analysis to find declarations of software entities inside a file. We used CTAGS to compute which methods exist within a file and where they start. However, CTAGS does not report where the entities end. Therefore, our tool had to process the output of CTAGS for each file to locate the finishing line of each method. Our tool sorts the entities defined in a file by the line of code in which they are defined. To find the end of a method, our tool looks for the matching closing brace, thereby assuming that the file is syntactically correct. It is common that files are syntactically correct because projects usually have as policy to commit changes to the repository only when they compile successfully. In case the file is not syntactically correct, the tool would find the first closing brace when searching backwards starting from the entity defined in the file after the method whose boundaries are being calculated. This is done for all the methods that changed in the application.

The example below shows the use of CTAGS to obtain the full signature of the methods and the line where they start. The input file has at each line the path to the files to analyze, i.e. to the files that were modified in that logical change.

```
ctags       –L inputFile.txt
            –f outputFile.txt
            --extra=+f
            --fields=+a+f+i+K+m+n+s+S+z --excmd=number
```

The output file contains on each line a declaration of an entity found in the files contained in the input file. The example below shows the set of items obtained per entity declared, in this case, the method `compare(Object,Object)` of the inner class `StringCompare` in the class `StandardUtilities`:

```
compare
C:\jedit\jEdit\org\gjt\sp\util\StandardUtilities.java
```

[15] Available at http://ctags.sourceforge.net

```
347;"
kind:method
line:347
class:StandardUtilities.StringCompare
access:public
signature:(Object obj1, Object obj2)
```

For each entity it shows its name, path to the file that declares it and the line where it is declared, type of code entity, the number of the line where it starts, the class where it is declared, the access type of the entity, and the parameters (in case it is a method). All this information is separated by tab characters and identified by keywords such as '*kind:*' to describe the type of entity, '*class:*' to state to which class belong the entity, '*signature:*' to list the parameters of the method, etc.

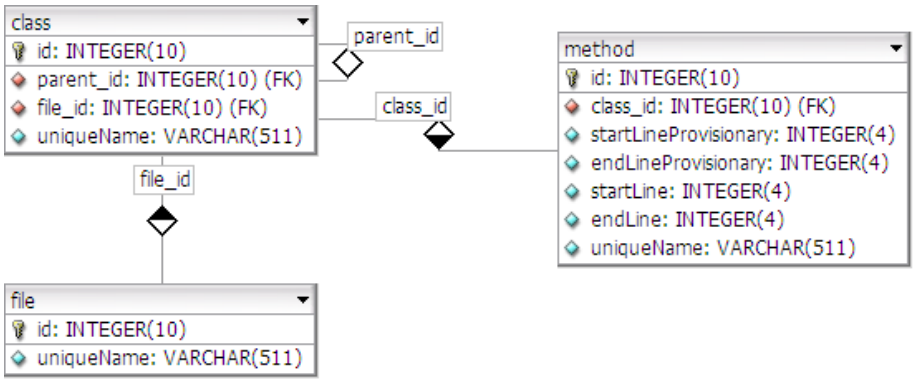


Figure 7-13. Tables that store the information about the methods, and about the software entities above them.

The information is stored into three tables of the database, shown in Figure 7-13. The table *file* stores a unique identification, and the path of each of the files that have composed the application. The table *class* saves for each class a unique identifier, the file where it is defined, its super-class, and its fully qualified name. The fully qualified name of a class is the name of the package in which the class is defined, and the name of the class, separated by a period. In case the class is an inner class, its fully qualified name would be the name of the package, plus a period, plus the name of the outer class, plus a separation character ('$'), plus the name of the inner class. In the case of the example above, the name of the class would be the name of the package, plus a period, plus `StandardUtilities$StringCompare`. The defined package per file is identified using the output of CTAGS. The table *method* stores a unique identifier, the class that defines the method, the signature of the method, the line in which the

method starts, and the line where the method finishes.

7.2.2.1 Tracking of methods across snapshots

The data collection algorithm searches the methods defined on each logical change. This means that the tracking of methods across snapshots must be done using their names. However, the names of methods change along their lifetime. In order to be able of calculate the lifetime of a method accurately, we store in a separate table the way in which methods are renamed across their lifetime. This table is called *method translation* (shown in Figure 7-14). The table stores a new identifier for each method (*id*), and assigns to it the identification corresponding method(s) found during data collection (*realMethod_id*).

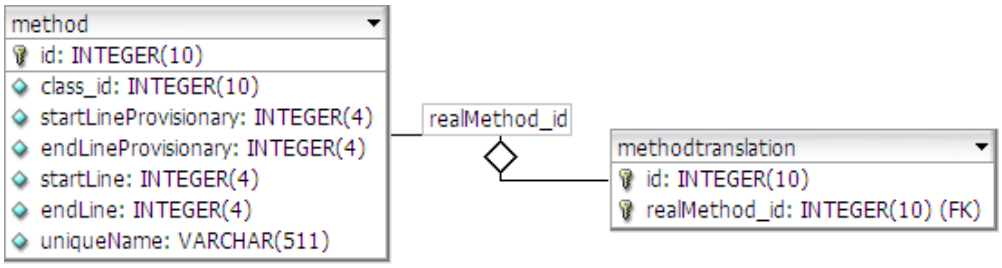


Figure 7-14. Tables that store the methods renamed reconstructed using origin analysis

In this way, the information stored about two methods, renamed versions of the same method, can be reconstructed before the analysis phases without adding complexity to the data collection algorithm. The rest of this section will cover the algorithm used to detect the previous version (i.e. the origin) of methods that seem new because they have a combination of name and location that was not stored before in the database.

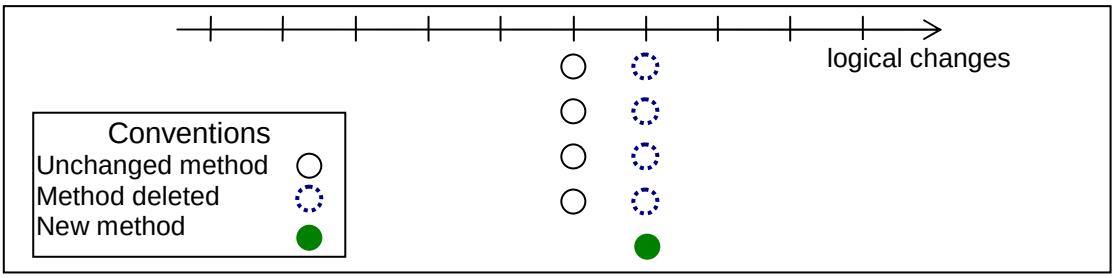


Figure 7-15. Origin analysis principle. The methods that are supposed to be new may be a renamed version of any of the methods that are supposed to be deleted.

As explained in the previous chapter, current approaches to find the origin of methods are not adequate for the analysis of applications over a large interval of time because they are costly to

calculate in terms of time. Therefore, we propose a new algorithm. As explained before, origin analysis algorithms aim to find the origin of a method that seems new, from the set of methods that seem deleted (i.e. the candidates). To decide if the 'new' method and the candidate are the same method, origin analysis techniques use two concepts: the identification of the methods, and the algorithm to compare such identifications. Our algorithm uses two identifications for a method: its complete name, and its content (LOCs). The name of the method is the identification used to filter most of the candidate methods to be the origin of the new one. The second identification (i.e. the contents of the method) is used only in case there is more than one candidate after the filtering done using the name of the methods.

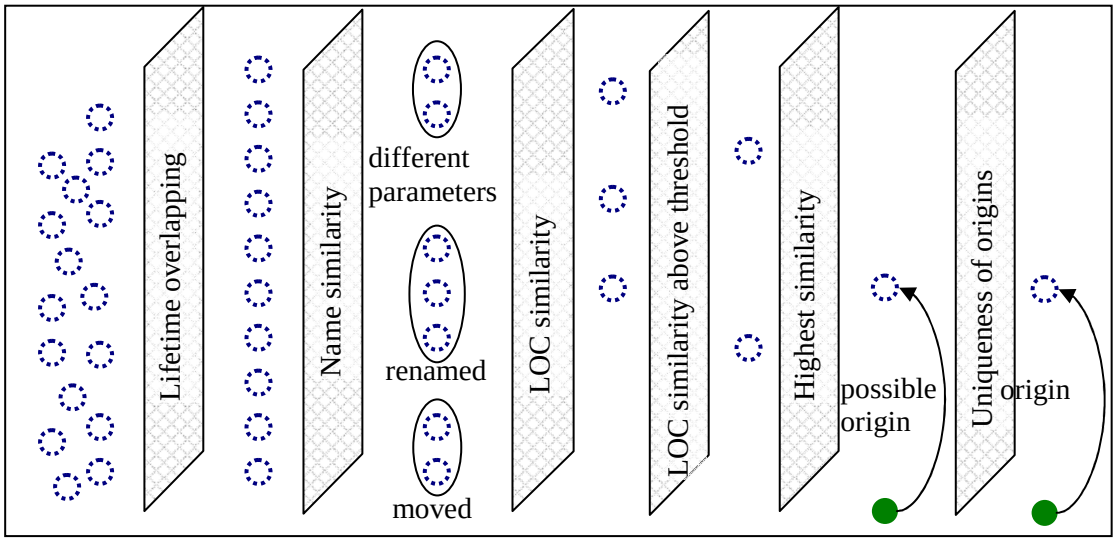


Figure 7-16. Our filtering of candidate methods (empty circles) to find the origin of the method that seems new (filled circle).

The first filter of candidates is done by eliminating those methods that at any point of their lifetime co-existed with the analyzed method[16]. This means that the first filter eliminates those candidates[17] that 'revived' while the method analyzed was alive. This happens when the methods are temporarily eliminated using comments. 'Revived' methods also occur when the file that hosts the method is accidentally deleted from the repository, being re-added afterwards. The second filter divides the candidate methods into three groups depending on their name. The

[16] Analyzed method: The one supposedly new.

[17] Candidate: Method that was deleted when the analyzed method was created.

identification of methods by the name considers the fully qualified name of the class and the signature of the method. The first group of candidates has the methods in the same class and with the same parameters but with different name (in Figure 7-16, the '*renamed*' set). The second group of candidates has the methods in the same class and with the same name but with different parameters (in Figure 7-16, the '*different parameters*' set). The third group of candidates has methods with the same signature located in a different class or package than the analyzed method (in Figure 7-16, the '*moved*' set). The third filter measures the similarity between the lines of code of the analyzed method and the candidates. Given that calculating this similarity may be costly in terms of time, only some of the sets of candidates are compared. By default, the candidates that are in the same class as the method analyzed (i.e. *renamed* candidates or *different parameters* candidates) have priority over the methods that seem moved. That means that similarity judged by the lines of code is not calculated for *moved* candidates. However, if the majority of the candidates belong to the same file or class, it is likely that the whole class or file was moved. In that case, *moved* candidates have priority, discarding the calculation of similarity by the lines of code for *renamed* candidates or *different parameters* candidates. The fourth filter eliminates the candidates whose LOCs similarity is below a threshold of 70%. In case there is no candidate remaining after this filter, the set of candidates that was not analyzed previously is compared for LOC similarity against the method analyzed. The fifth filter takes the candidates with the highest LOC similarity. In case more than one method has the highest similarity, the algorithm assumes that the method does not have a clear origin and marks it as a new method. It is assumed that the method analyzed is a new method cloned from those candidate methods that resemble it. Finally, the last filter checks that the origin of every method is unique, i.e. that two different methods do not have the same origin. This means that there are cases in which two methods apparently created at the same time are very similar to a third origin 'candidate' method that was deleted at the same time. Note that this happens when a method is divided into two methods, each one with a different signature than the initial method. This is a common refactoring on large methods, with several parameters, which handle multiple responsibilities. As explained before, in these cases, the origin is assigned to the new method that has higher similarity with the deleted method. In case of having equal similarity between the 'new' methods and the origin, the algorithm assumes that the origin was a method deleted and that the 'new' methods are indeed new.

The similarity by lines of code works as follows: the shorter (i.e. the one with fewer Lines Of

Code) of the method and the candidate is detected. For each line in the shorter method, the algorithm finds the line that resembles it most from the lines that compose the larger method. Then the similarity between the two methods is the sum of the similarities found for all the lines of the shorter method, over the number of lines in the shorter method.

The similarity between two lines of code is calculated as the average of the percentage of characters that have in common parallel tokens in the lines. The percentage of characters in common between two tokens is the number of pairs of consecutive characters that the tokens have in common (counted twice), over the characters of letters of both tokens. Note that the common characters are counted twice in order to obtain one when comparing the similarity on equal strings. The tokens are recognized by taking into account the special characters in the language such as spaces, operators, braces, semicolons, etc. Nevertheless, tokens may also be divided into words whenever there is a change of capitalization as in `taxesCalculation`, or when there are intermediate characters like in `taxes_calculation`. The separation of tokens into words permits to increase the accuracy on the similarity of the semantics between the analyzed lines.

The results of the extraction of logical changes are shown in Table 7-3.

Table 7-3. Summary of methods identified in the analyzed applications

Application	Number of methods detected initially	Number of methods after origin analysis
Freecol	4099	4050
JEdit	8434	8004
Ganttproject	14895	14616
Columba	28876	28376
JBoss mod.	12144	12132

7.2.3 Identification of clones

Giesecke pointed out desirable characteristics from clone detection tools, which include: being language independent, being independent of the detection approach, and being able to detect clones at different levels of similarity [Giesecke '07]. The first characteristic is desirable because it permits analyzing applications written in different languages, which would permit comparing the impact of the programming language in the types of clones that an application may have. Besides, a language independent clone detector allows the analysis of a wider range

of applications. The last two characteristics permit deciding which clone instances are false positives.

CCFinder is an automatic clone-detection tool that uses lexical analysis to normalize the source code, which is then transformed by language dependent rules into a sequence of tokens. Finally, a string based comparison between the tokens locates the clones [Kamiya '02]. We decided to use CCFinder as clone detection tool for several reasons. First, it would allow us to compare our results with many empirical studies on cloning ([Monden '02; Ueda '02; Kapser '03; Kapser '04; Kim '05; Geiger '06; Kapser '06a; Kapser '06b]). Second, because CCFinder is capable of detecting three of the four types of clones in terms of similarity level (see in this chapter section ‘Classifications of clones’ on page 51). Third, because CCFinder has one of the best recall levels, while still keeping a reasonable precision, from the tools that have been used for large scale analyses [Bellon '07] (see section ‘Advantages and disadvantages of each code representation’ on page 44, and ‘Comparison of the most popular clone detection tools’ on page 50). The fact that it produces data with high recall and reasonable precision permits to have a rich dataset that can be filtered if desired, as Giesecke suggests [Giesecke '07]. Finally, given that CCFinder is based on token comparison, its time performance is very good, which is an important requirement when analyzing the history of clones.

The data collection algorithm parses CCFinder's output. Although CCFinder can detect clones on different programming languages, one of the intermediate files produced by CCFinder that needs to be parsed depends on the programming language; therefore, our algorithm only handles Java applications.

CCFinder was configured to find code clones with a minimal length of 30 tokens, with tokens of at most 10 characters, to distinguish different identifiers, and to ignore block structures so that clones are not partitioned, as shown in the example below.

```
ccfx.exe d  -i inputFile.files
            -o outputFile.ccfxd -b 30 -t 10 –v
```

The size of the token was chosen because it is the default of CCFinder, probably for performance reasons. However, this does not mean that similarity in larger tokens is dismissed. In case the token is larger than 10 characters, the token is divided into several tokens of the same type that are compared sequentially. Therefore, identical tokens would be recognized

anyway.

The output of CCFinder is a file containing the clone tokens that form the clone relations and the clone families, and an intermediate file for each source code file analyzed that stores the translation of the file to tokens. An example of the output of CCFinder is presented below:

```
source_files {
1     C:\client\ClientOptions.java  1587
2     C:\client\control\ClientModelController.java    422
...
}
clone_pairs {
7           1.4-43            1.40-79
918         1.601-667         1.1211-1277
1039        1.739-770         1.780-811
1039        1.739-770         286.95-126
...
}
```

The first set of lines indicates each file: its identification, its full path, and its number of tokens. The second set of lines describes all clone relations in the application in three columns. The first column contains a unique identifier per clone family, it indicates that the clone relation in the following two columns belong to that family. The second and third columns specify the fragments that compose the clone relation: the first number indicates the identifier of the file in which the fragment is located, and the two following numbers indicate the first and last tokens that belong to the fragment. For instance, the first line of the clone relations says that there is a clone relation belonging to the family identified as 7, the first fragment goes from the token 4 to the token 43 of the file 1, and the second fragment goes from the token 40 to the token 79 of the file 1.

It is necessary to translate the outcome of CCFinder to lines cloned within each method and to which clone family those lines belong. Therefore, it is necessary to know the conversion of files to tokens, which is contained in the intermediate files produced by CCFinder. An example of such files is shown below. Each line in the intermediate files represents a token. The first column indicates the position of the token, the second column indicates its length, and the third column indicates the type of token. Three hexadecimal numbers separated by periods describe the position of the token. The first number indicates the line of the file in which the token is (starting with line 1), the second number indicates the column in which the token begins

(starting with column 1), and the third number indicates in which character the token start counting from the first character in the file and starting with character zero.

```
6.8.41      +0    (def_block
6.8.41      +5    r_class
6.e.47      +8    id|ListSet
6.16.4f     +1    (brace
9.9.88      +7    id|ListSet
9.10.8f     +1    (paren
9.11.90     +1    )paren
9.13.92     +1    (brace
a.3.97      +5    id|super
a.8.9c      +1    (paren
a.9.9d      +1    )paren
a.a.9e      +1    ;
...
```

The lines of code described by the token description above are:

```
   ...
6  public class ListSet {
7     private Vector myElements = new Vector();
8
9     public ListSet() {
10          super();
11    }
   ...
```

Notice that CCFinder does not translate all the statements in the source code e.g. none of the elements in line 7 have a token identification. Furthermore, note that modifiers are not taken into account in the translation. The mapping from tokens to methods is done using the lines of code that belong to each method, which are updated after each logical change, and stored in the table that stores the changes per methods (*methodChange*). The tokens are stored in the columns *startToken* and *endToken*, the rest of the values in the row are filled with the identifier of the method (*method_id*), and of the logical change (*commitTransaction_id*), and default/empty information for the rest of columns (see Figure 7-20 in next section).

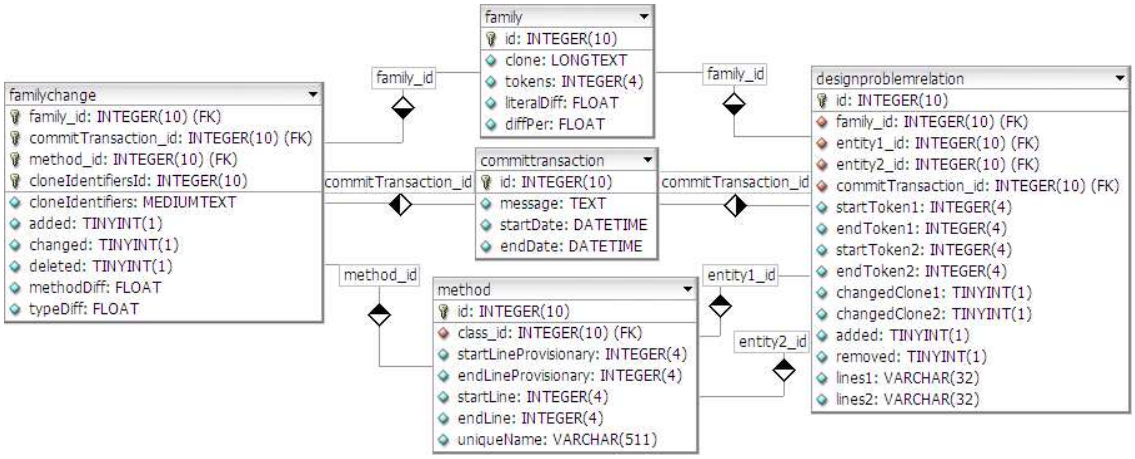


Figure 7-17. Tables that store clone relations between methods, and their corresponding clone families

7.2.3.1 Storing clones detected

Every time that a file is affected by a logical change, its clones must be recalculated. The data collection algorithm runs CCFinder over the whole snapshot but our tool only processes those cloned fragments in which the changed files are involved. For each clone-relation with more than 30 tokens inside the methods involved in the relation, the algorithm verifies if the cloned fragment belongs to any of the clone families stored. This is done by calculating the clone representing the clone relation (see Figure 5-6 on section 5.2.1). The calculation of the clone for a clone-relation is done in two steps: first, reconstructing the cloned fragment, and second, replacing the tokens that differ between all cloned fragments in the clone-family by special characters. Reconstructing the cloned fragment is done by finding the tokens inside one of the methods, and transforming it into a normalized syntax i.e. similar indentation, braces placing, etc. Replacing the tokens that differ is done by finding all the cloned fragments that belong to the clone family detected by CCFinder in that logical change. Afterwards, the index of the tokens that differ in any of the fragments is stored. Finally, the tokens that differ are replaced in the normalized code fragment by special characters using the indexes previously stored. The result is the clone that represents the family of the clone-relation found. The clone is compared against the clones stored in previous logical changes that are in the family table (see Figure 7-17). If the clone is found in the table family, the clone relation formed by the fragments is stored in the table *designProblemRelation* (in Figure 7-17). The history of the family is also checked to verify if the cloned fragment was already in the family or not.

7.2.3.2 Storing clones history

The history of a clone family is stored in the table *family change*; each entry in the table indicates when each cloned fragment of each family was added, removed, or changed. If there is no entry indicating that the fragment in that method belongs to the family identified (see above), a new entry is created, marking the addition of the cloned fragment to the family. If the cloned fragment belongs to the clone-family, it is necessary to identify if the cloned fragment was changed. Given that the cloned fragment still complies with the clone that defines the family, the only way its change can be detected is by identifying if any of the tokens that are different among the fragments of the class changed. The different tokens per fragment are stored in the table *family change*, in the column *cloneIdentifiers* that stores a string with the tokens that differ separated by commas, and in the column *cloneIdentifiersId* that contains the hash value of the identifiers. The column *cloneIdentifiersId* is part of the key of the table *family change* because there could be several fragments of a family in the same method. The lines of the fragments are not used to differentiate clone fragments of the same method in the same family because numbers of the lines are likely to change every time the file is modified, due to additions or deletions of code before the fragment; being therefore, an inefficient approach. Therefore, in order to differentiate clone fragments of the same method in the same family we use the identifiers that replace the special characters in the clone to obtain the cloned fragment (an example of such identifiers is shown in Figure 5-7). The fact that identifiers of the cloned fragment are part of the key of the table *family change* implies that the changes in the family must be identified with a post-processing algorithm.

The results of the extraction of clones are shown in Table 7-4.

Table 7-4. Summary of clones identified in the analyzed applications

Application	Number of clone-families	Number of clone-relations
Freecol	1159	1407
JEdit	1289	1454
Ganttproject	1728	2100
Columba	7340	10870
JBoss mod.	2649	3787

7.2.4 Identification of changes

Tracking methods and clones across time is not enough to collect the data required for analysis. Because the methodology aims to assess the effect of SCIs on changeability, it is necessary to track the changes, identifying which methods and clones are affected. SCM systems have operations that provide the number of the lines changed from one version to another version of the same file. The number of the lines added or modified in a logical change corresponds to the version of the file after the logical change. The number of the lines deleted in a logical change corresponds to the version of the file before the logical change. Consider the example presented in Figure 7-18.

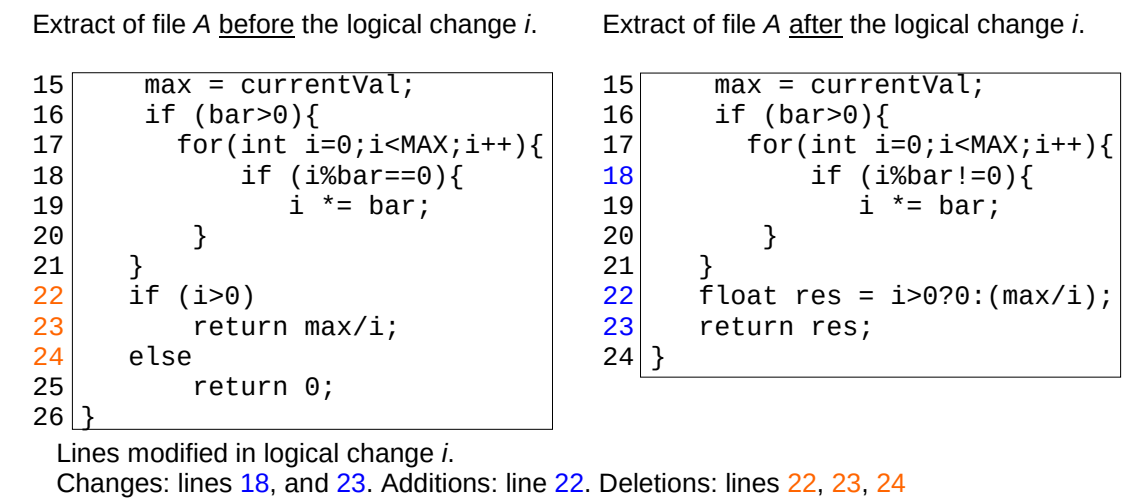


Figure 7-18. Storage of physical changes occurred in a logical change

In order to find the methods or cloned fragments modified in a logical change, it is necessary to know which lines compose each method and each cloned fragment. The lines that compose each method and each cloned fragment can be extracted when they are located. To know if a method or a fragment was modified by a logical change is done in two steps. The first step is to check if any of the changed or added lines correspond to the lines of the method *after* the logical change. The second step is to check if any of the deleted lines correspond to the lines of the method *before* the logical change, and similarly for cloned fragments.

Given that the lines composing each method and each clone at any snapshot are known, identifying changes inside methods or clones only requires finding the lines added, changed, or

deleted by the logical change. CVS systems permit knowing the changes in the lines of a file from one version to the next. This information is provided by CVS through the *diff* command.

The data collection algorithm calls the diff command for each file changed, for the initial timestamp of the previous logical change in which the file was modified, and for the final timestamp of the logical change whose data is being gathered (see the example below). Calling the diff command with these parameters would compare these two versions of the file in the CVS repository.

```
diff        -c  -w  –D 04/06/2006 17:53:32 GMT
            –D 30/12/2006 01:34:31 GMT
            project/Example.java
```

The output of *diff* is the list of the changes done to the file between the versions analyzed. The lines corresponding to each version are differentiated in the diff output by different characters. The example of diff output shown on Figure 7-19, shows the lines of version 1.7 identified with asterisks and the lines of version 1.8 identified with hyphens. For each change, the diff output shows the context of the change for both versions of the file, that is, three lines before and after each change. It is necessary to know the lines that compose each method in both versions analyzed. Note that *changed lines* can be identified using the limits of the method in any of the versions analyzed, *added lines* can only be identified with the limits of the method in the latter version analyzed, and *removed lines* can only be identified with the limits of the method in the earlier version analyzed. That is the reason for having two start and end lines in the table *method* (in Figure 7-13), so that at every logical change the lines of the previous version are stored in the provisional columns. The lines of the current version can be stored in the columns designed to hold the current lines that compose the method, i.e. in the column *numOfLines* of the table *methodChange* see Figure 7-20.

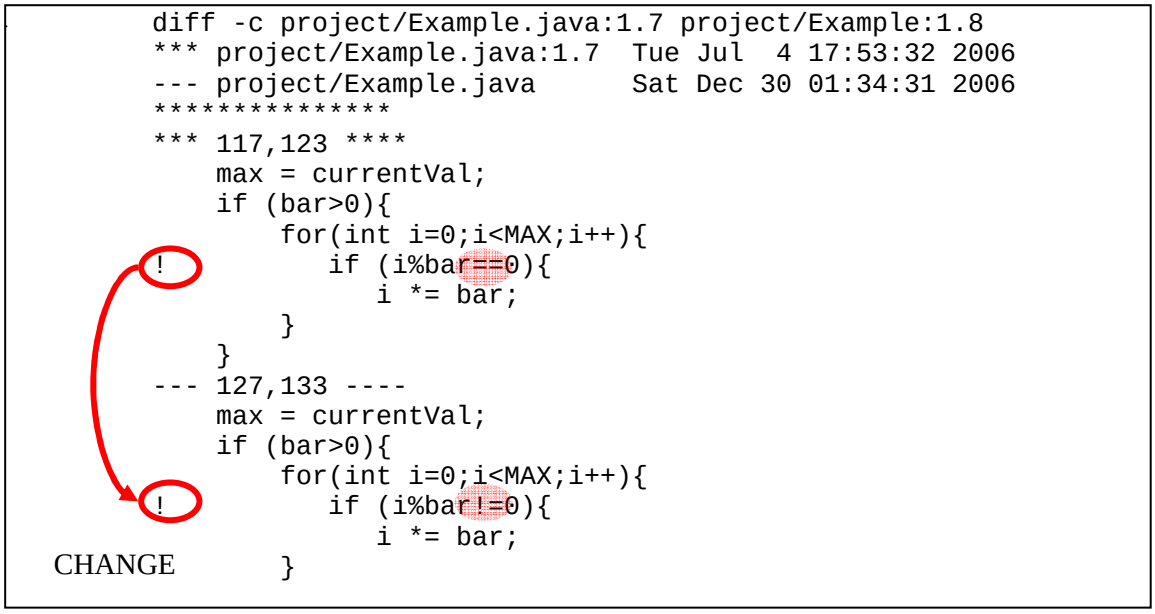


Figure 7-19 CVS Diff example.

Whenever a file is identified as deleted all the methods defined on it are marked as deleted, and the deletion is propagated to the clone relations (i.e. table *design problem relation*), and to the clone classes (i.e. table *family change*).

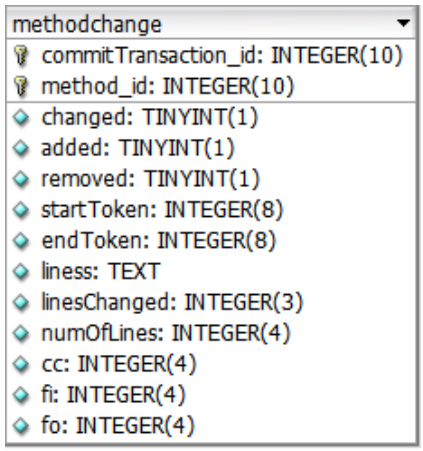


Figure 7-20. Table that store changes in methods

Once the changes per method are identified, they are stored in the table *method change* (see Figure 7-20). The table indicates if the method was added, removed, or changed at a given logical change. For each change in a method, the table stores:

→ *the number of lines changed in that method at that logical change (in the column*

numOfLines)

→ *the number of lines changed in the column linesChanged*

→ *the changes in its lines as they appear in the log file in the column liness*

→ *the type of changes in the columns added, removed, and changed*

→ *and the tokens that compose the method at that moment in time (in the columns startToken and endToken).*

Besides, this table also keeps a log of characteristics of the method in columns *cc*, *fi*, and *fo*. These characteristics are discussed later in section 7.2.5.

Changes inside cloned fragments are located by identifying if the lines changed on each method were the same lines that were cloned. Given that the table *design problem relation* saves not only the methods involved on each clone-relation, but also the lines corresponding to the cloned fragment in those methods, finding the changed clones it is enough to propagate the information of the lines changed per method. The information of changes in clones is stored in the table *design problem relation*, in the column *changedClone1* if the first fragment stored in the relation was modified, and in the column *changedClone2* if the second fragment stored in the relation was modified (see Figure 7-17). Note that the changes inside cloned fragments cannot be stored in the table *family change*, because that table stores changes of the clone-family, but not all the changes in a cloned-fragment changes the clone family (see the types of changes in clone families in section 3.4). For instance, renaming a variable in a method could result in changing one of the tokens that change in the clone that represents the family. However, if the rename is done using a refactoring tool this change in the clone fragment does not affect the clone-family because it does not affect in any way the clone that represents it.

7.2.5 Identification of attributes

Calculating how the logical changes affected the methods and their cloned fragments permit us to implement the majority of the analysis phases in the methodology. However, characterizing methods over time is also necessary because the changeability of methods in the periods cloned and not cloned may differ due to characteristics of the method rather than to the fact of having cloned fragments. Moreover, characterizing cloned fragments over time is also necessary because, in case the characteristics in the method do not explain differences in changeability, the characteristics of cloned fragments may help to identify what makes a clone harmful for the changeability of the method where it is placed.

The tools that calculate characteristics of methods and clones should comply with the same characteristics as the tools for identifying the structure of SCEs. Those characteristics are: being based on the processing of source code files, being independent of changes in the grammar of the programming language (i.e. cannot be based on syntax analysis), having an output which is easy to process, be lightweight, and gathering as many of the attributes required as possible.

7.2.5.1 Clones' characterization

The algorithm of data collection does not require a third party tool to calculate the characteristics of clones, given that they are extracted when the cloned fragments and the clone families are stored in the database. The characteristics to analyze per clone are summarized in Table 7-5.

Table 7-5. Characteristics to analyze that are relative to the clone in the method

Characteristic	Description
Characteristics relative to the clone	
Size	Number of tokens per fragment
Characteristics relative to the clone family	
Family size	Number of fragments that compose the family
Similarity to clone	Percentage of tokens that are different between the cloned fragment and the clone that identifies the family[18]
Method call / Type similarity	Percentage of differences in tokens referring to method calls and types between the cloned fragment and the clone that identifies the family
Literal percentage	Percentage of tokens that refer to literals in the clone that identifies the family
Characteristics relative to the relation method-clone	
Lifetime affected	Percentage of the method's life in which it has a fragment cloned
Percentage affected	Percentage of tokens cloned from the tokens of the method
Changes in clones	Number of changes inside the cloned fragments

[18] The clone that identifies a family is the cloned fragment with a normalized syntax (i.e. indentation, bracket placement, etc.) with special characters to represent the tokens that differ on the cloned-fragments. See an example in Figure 5-6.

7.2.5.2 Methods' characterization

The algorithm of data collection uses SPOON[19] [Pawlak '06] to calculate some of the characteristics of methods. SPOON is a syntactical analyzer of Java code used also for source-code-to-source-code transformations. A lighter version of SPOON that analyzes code regardless of its syntactic correctness was developed for the purpose of this work. This version of SPOON uses an extension of Java annotations to mark the Java code as its model is traversed. The marks are collected afterwards to calculate metrics about the source code. Given that the attributes of methods may change over time, they are stored in the table that stores changes in methods (i.e., table *method change*), having a column per characteristic to save. The characteristics of methods to store are summarized in Table 7-6.

The characteristics of methods to store are its Lines Of Code (column *numOfLines*), its cyclomatic complexity (column *cc*), its fan-in i.e. number of methods that call the method analyzed (column *fi*), its fan-out i.e. number of methods that the method analyzed calls (column *fo*). These characteristics are stored in the table that stores method changes (see Figure 7-20), in the columns *numOfLines* for the Lines Of Code, *cc* for the cyclomatic complexity, *fi* for the fan-in, and *fo* for the fan-out. The number of parameters and the age of the method are calculated based on the information collected for gathering the history of the application; that is, the name of the method and its changes.

Table 7-6. Method characteristics to analyze.

Characteristic	Description
LOC	Lines of code of the method
Complexity	Number of branches of the method
Fanin	Number of methods that call the method
Fanout	Number of methods called by the method
NOP	Number of parameters
Age	Number of logical changes that the method has been part of the application

[19] Download from: http://spoon.gforge.inria.fr.

7.3 Summary

This chapter describes how to collect the information required for applying the whole methodology to clones. The data collection process presented should be followed in a strict order. Instead of presenting the technical challenges related to this phase, which are discussed section 2.3.1 based on related literature, this chapter presents the best alternatives found. This chapter also presents the data collection algorithms and persistence schema that we developed in order to apply the methodology to clones at method level. The contributions of this chapter are a novel way to detect origin analysis, and a new tracking mechanism for the evolution of clone families by storing not only the location of the cloned fragments but also the clone that represents the common snippet in the family.

The following chapter explains how to identify typical values on the characteristics of SCIUS, and the results when applied to analyze clones.

Chapter 8. Nature of the Source Code Issue Under Study

This section explains how to find similarities among instances of the SCIUS to understand better the effects of SCIUS. The following section describes how to evaluate the evolution of the SCIUS in terms of changeability.

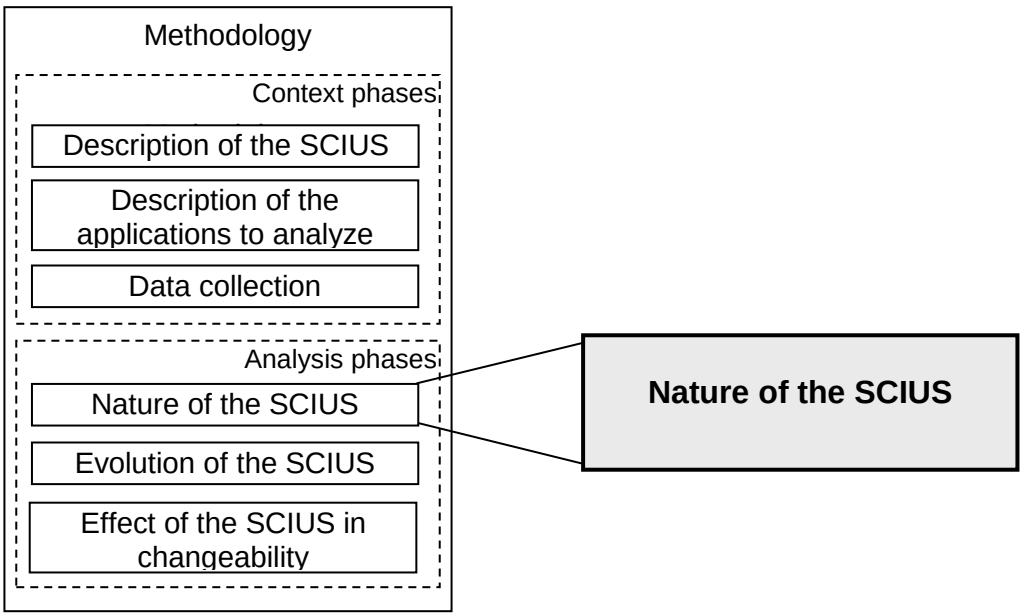


Figure 8-1. First analysis phase of the methodology. Nature of the SCIUS.

8.1 Phase description

This section presents the description of the phase "Nature of the SCIUS", and the steps proposed to achieve the phase.

Deliverable: Typical values for characteristics of the SCIUS, and the implications of those characteristics on the changeability effects generated by the SCIUS.

Rationale: This analysis is exploratory. This means that the phase aims to reduce the search space for relations between the SCIUS and the changeability of the SCEs. Nevertheless, this phase does not aim to produce quantitative results, but rather, to indicate which characteristics

may be related with the effect of the SCIUS on the changeability of the SCEs. The purpose of this analysis is to discover interesting characteristics about the SCIUS by finding common values in the majority of the SCIUS instances. Visualization tools are not only a support for foraging data into information, but also as a sense-making mechanism that permits finding insight out of information [Yi '08]. Given that, the human eye is able to recognize effortlessly the predominant color in a figure [Ducasse '07], the analysis proposes to show different characteristics related to the SCIUS as colors on figures that represent SCEs.

Procedure: From the set of characteristics that have not been used to classify the SCIUS, select those that may have been related with the changeability of the SCE. Explain for each characteristic selected its relation with the changeability of the SCE. For each characteristic, describe how the changeability could be altered for values of the characteristic. Draw a figure for each SCE with the SCIUS analyzed, and fill each figure with a different color for each value that a given characteristic can take. Report the most common and the least common values.

8.2 Phase application

As mentioned in the previous chapter, although there have been many descriptions of cloning, there are still several characteristics that have not been reported. In this section, we describe the clones on the case studies based on the following characteristics: their creator, eliminator, their size, the percentage of the method affected, the lifetime of the method affected, the location of the clones, the maintainer of the clone, and the level of similarity of clones.

To analyze these characteristics we have developed an application that builds a graph where the nodes are the methods that have had clones, and the edges are the clone relations among methods. This graph is depicted by GUESS[20]. The nodes have the characteristics of the cloned fragments inside the methods. We have also developed a script for GUESS to highlight the methods in different size and colors according to the value of their cloning characteristics. The details on how the script behaves depending on each characteristic are described below.

In order to analyze the nature of clones in the application, that is describing the majority of the clones, we have developed an application that converts part of the cloning information stored in

[20] Available at: http://graphexploration.cond.org/

the database in a GUESS graph. An example of the graphs obtained is shown in Figure 8-2. Each dot represents a method that has had a cloned fragment at any moment of its lifetime, while the edges represent clone relations between the methods, i.e. cloned fragments shared. Note that for two methods to belong to the same family; it is not enough to share a relation with a third method, it is necessary to have an edge to it. The colors of the dots have a meaning, which changes depending on the characteristic analyzed. The colors permit the identification of characteristics of clones. The graph gives information regarding the families because each method is connected by an edge to each one of the methods with which it had a cloning relation. One of the layout options of GUESS (GEM) locates the large and complex sets of relations (i.e. large clone-families) in the center of the graph, and the small and simple relations (i.e. small clone families) in the periphery. In this way, it is easy to identify if there is any relation between the characteristic analyzed and the type of clone family.

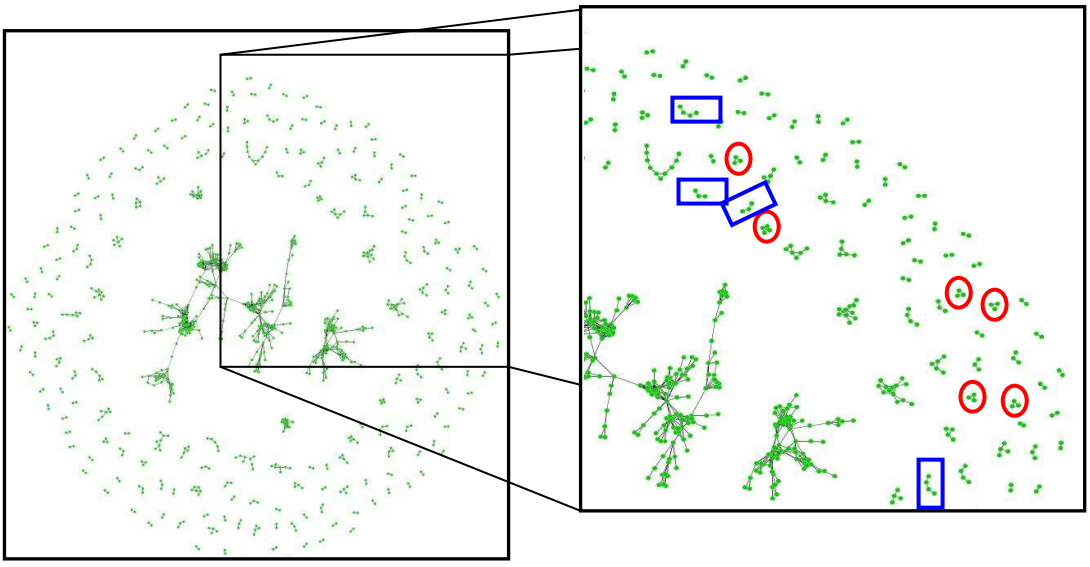


Figure 8-2. Example of cloned methods per application. Each dot is a method; each edge is a clone relation.

Figure 8-2 shows the cloned methods in Freecol. Note that most of the families have few members, in many cases just two, as most of the families are located in the borders and are pairs of dots. Note also that in many occasions one method can be related with many other methods but that they do not necessarily belong to the same family, those methods are usually located in lines (see the rectangles in the figure). Methods belonging to the same family are all cloned among themselves, i.e. in a blackberry shape; see in the figure the sets of methods in a circle.

Closer methods tend to be methods that share a clone relation. The analysis of clone characteristics requires coloring the dots of the graph, is done, with a GUESS-script we developed.

The characteristics that we propose to analyze the nature of clones are related to the cloned fragments inside a method, to the relation between the method and its cloned fragments, and to the clone families inside a method. The characteristics related to the cloned fragments are:

- the creator,
- the commit of creation,
- the eliminator,
- the commit of elimination,
- the size of the cloned fragments in a method,
- and the role of the cloned fragment when cloned.

The characteristics associated with the relation between the method and its cloned fragments are:

- the relation between the owner of the method and the owner of the clone,
- the percentage of the method affected by cloning,
- the percentage of the method's lifetime affected by cloning,
- the full method name of the methods that belong to the same clone family,
- and the age of the method when cloned.
- The characteristics related with the clone family are:
- the family dissimilarity,
- the percentage of literals,
- and the percentage of tokens that refer to method calls or types that differ.

Several of the characteristics analyzed in this chapter have been previously tackled in the literature (see Figure 8-3). However, previous studies may not accommodate to the settings defined to apply the methodology, that is, clones at the level of methods. For instance there are not studies that say the distribution of:

- number of tokens cloned in a method. However, there have been several studies analyzing the number of lines of code per cloned fragment (see size of fragment in section 3.3.3).
- percentage of tokens cloned in a method. Nevertheless, there are several results on the percentage of the application affected by cloning. Therefore, the percentage is usually measured in terms of lines of code, files, or subsystems cloned (see percentage affected in section 3.3.2).
- percentage of the lifetime of a method in which it is cloned. There is only one experiment regarding the lifetime of clones; in this experiment, the lifetime of a clone fragment is analyzed with respect to its clone family (see lifetime in section 3.3.1). Therefore, if the clone mutates into a new clone it is considered volatile.
- age of a method when it becomes cloned. In [Monden '02], there are three categories of modules: young (within 4000 days of being created), middle (between 4001 and 8000 days of being created), and old (more than 8001 days of being created). They found that most of the clones were located in young modules (see age in section 3.3.2). However, we think that the definition of age is too coarse and that it might be useful to repeat the analysis between age and clones with a more detailed definition of age.
- percentage of dissimilarity of a clone fragment with respect to its family. Although the metric was proposed before, it was used to discriminate clones from false positives (see similarity to clone in section 3.3.4). Therefore, the only information we know from the authors that proposed it is that between 3% and 37% of the clones they found had at least 50% of dissimilarity with the clone that represents its family. However, the dissimilarity may indicate types of clones: lexical, semantic, or structural. Therefore, the dissimilarity to the clone may be useful when analyzing the effect of a clone on the changeability of the method that hosts it.
- or percentage of method-type. This characteristic has the same problem of the percentage of dissimilarity to the clone i.e. having been used only to eliminate false positives, when it can be useful to distinguish types of clones (see method call similarity in section 3.3.4).

Analyzing the characteristics listed previously, with other metrics would allow us confirming previous results or pointing out variations depending on the interpretation of the characteristic.

Characteristics analyzed in the literature (see nature of clones on Figure 5-8)		Characteristics analyzed in this chapter
		Creator
Scope	Age	Eliminator
Distance	Clone size	Elimination
Intention	Percentage cloned	Ownership
Wildcards	Lifetime cloned	Name
Similarity level	Dissimilarity w.r.t. Clone	Role
Family size	%method-type differences	Creation
		%Literals

Figure 8-3. Characteristics analyzed in the literature vs. characteristics analyzed in this chapter

The rest of characteristics analyzed are new for describing clones. Creator, and ownership would allow us to explore to whether or not certain developers tend to clone more. This is interesting because some authors suggest that programmers tend to clone when they are learning how the application works. However, this contradicts the fact that a developer must know the code in order to identify which parts implement a similar functionality. The eliminator characteristic would show if specific developers tend to refactor clones or not. The creation and elimination characteristics would indicate if there are periods in which clones are massively created or destroyed. The name of the methods would allow us to check the semantic and structural distance between methods whose clones belong to the same family. The role would indicate to what extent clones are created by copy-and-paste. Finally, the percentage of literals would indicate to what extent clones are just accidental similarities in the literals used. Notice that, although having the same literals would indicate some level of semantic relation between methods, this relation does not imply that they require parallel maintenance.

The sub-sections below explain each one of these characteristics, the expected outcome, and the results obtained from analyzing the graphs.

8.2.1 Creator

The **creator** characteristic refers to the developer that created the first clone relation of a method. This characteristic would show us if only certain developers are responsible of the cloning in the application, or if it is a generalized practice. However, before concluding that certain developers are responsible for the cloning in the application one should verify that such developers are not the ones that always handle those methods. For instance, D'Ambros et al. have found that in OSS application there is usually a main developer in charge of the majority of the code, if this developer is the one responsible for the cloning in the application, it might be

because the other developers do not contribute enough to the application to show their cloning [D'Ambros '08]. Another interesting fact to analyze about the creator of the clones is to find if the cloned fragments of the same family are introduced by the same developer or not. Balint et al. have found that there are cases where clone families are created by the same developer, and cases where several developers are involved [Balint '06]. However, they did not investigate which case was more common.

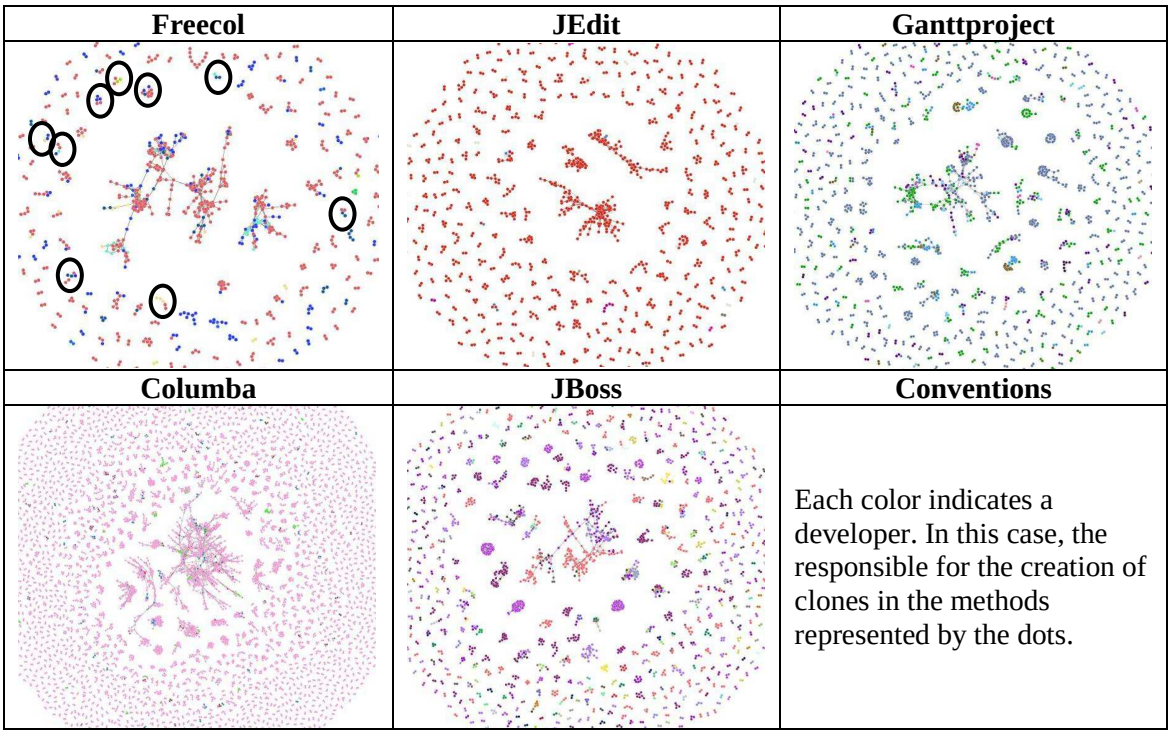


Figure 8-4. Creator of clones.

Clones graphs on Figure 8-4, where each color represents a different developer, and the color of each point indicates that developer was responsible for introducing the first cloned fragment in that method. We have found that applications with a small number of developers seem to be cloned by the same person. However, when looking at applications with a larger set of developers (like JBoss) it is evident that more developers participate in the creation of clones. The applications are different because some applications tend to have a single main contributor, while other applications have a more even distribution of source code contribution across developers. In general, the relation between developers and clones is that the more a developer contributes to an application, the more likely it is for her to introduce clones. Freecol has 14 developers but three main contributors that are responsible for 44%, 24%, and 15% of the commits. JEdit has 13 developers but one main contributor that committed 82% of the changes.

Ganttproject has 20 developers, with 2 main contributors responsible or 48% and 37% of the commits of the application. Although Columba is a large application with 16 developers, from which three are main contributors, in charge of 48%, 19%, and 18% of the commits. The only application that presented an even distribution of the commits across developers is JBoss, with 86 developers and an average of 1% commits per developer. The largest contributor to JBoss committed only 13% of the changes. Finally, note that although the same developer creates most of the families (i.e. clusters of dots), there are several examples of different developers creating cloned fragments of the same family like those proposed in [Balint '06]. We have marked some noticeable examples in Freecol with black ovals enclosing those clone families.

We examined these intuitions by comparing the percentage of commits done per developer for those developers that tended to insert the first cloned fragment in the methods of the application (see Table 8-1). We have found that although there is a relation between the contribution of the developer to the code-base and the contribution of the developer to clone creation, this relation is not very strong. Several developers with an important contribution to the clones of the application are not among those developers that contributed the most to the code-base of the application (which are highlighted in Table 8-1 with bold). Besides, several developers that contributed significantly to the code-base did not added clones in a similar proportion; these developers are the shadowed cells in Table 8-1. Therefore, it seems that cloning is a generalized practice among developers. However, the contribution to the code-base is not a reliable predictor of the contribution to clone creation.

Table 8-1. Developers that cloned the highest amount of methods (in decreasing order), compared with the percentage of commits done by developer.

Freecol		JEdit		Ganttproject		Columba		JBoss	
clones	commits	clones	commits	clones	commits	clones	commits	clones	commits
69%	44%	95%	82%	56%	48%	92%	48%	17%	13%
17%	24%	3%	6%	18%	37%	3%	19%	**16%**	**3%**
6%	**2%**	1%	4%	9%	6%	2%	18%	**15%**	**4%**
3%	3%	1%	2%	**6%**	**0.3%**	1%	3%	**12%**	**2%**
2%	15%	0.4%	2%	3%	2%	1%	3%	8%	7%

8.2.2 Creation

The **creation** characteristic refers to the logical change/commit transaction in which the first clone relation is added to a method. This characteristic would indicate if there are hot-spots on clone creation along the history of the application, and if the fragments of a clone family are

usually created at once, or if they are created in different moments.

We have found that there are no hot spots of clone creation in the history of the applications. In general, methods are cloned as the application ages. These patterns can be identified in Figure 8-5. Blue-like colors indicate that the first cloned fragment was introduced in the method in the earlier commits, while red-like colors indicate that the first cloned fragment in that method was introduced in the latter commits of the application. Note that the most large and intricate clone families are introduced at the beginning of the application's history, having several 'newer' additions. This indicates that large clone families are long-lived, and that they keep growing over time.

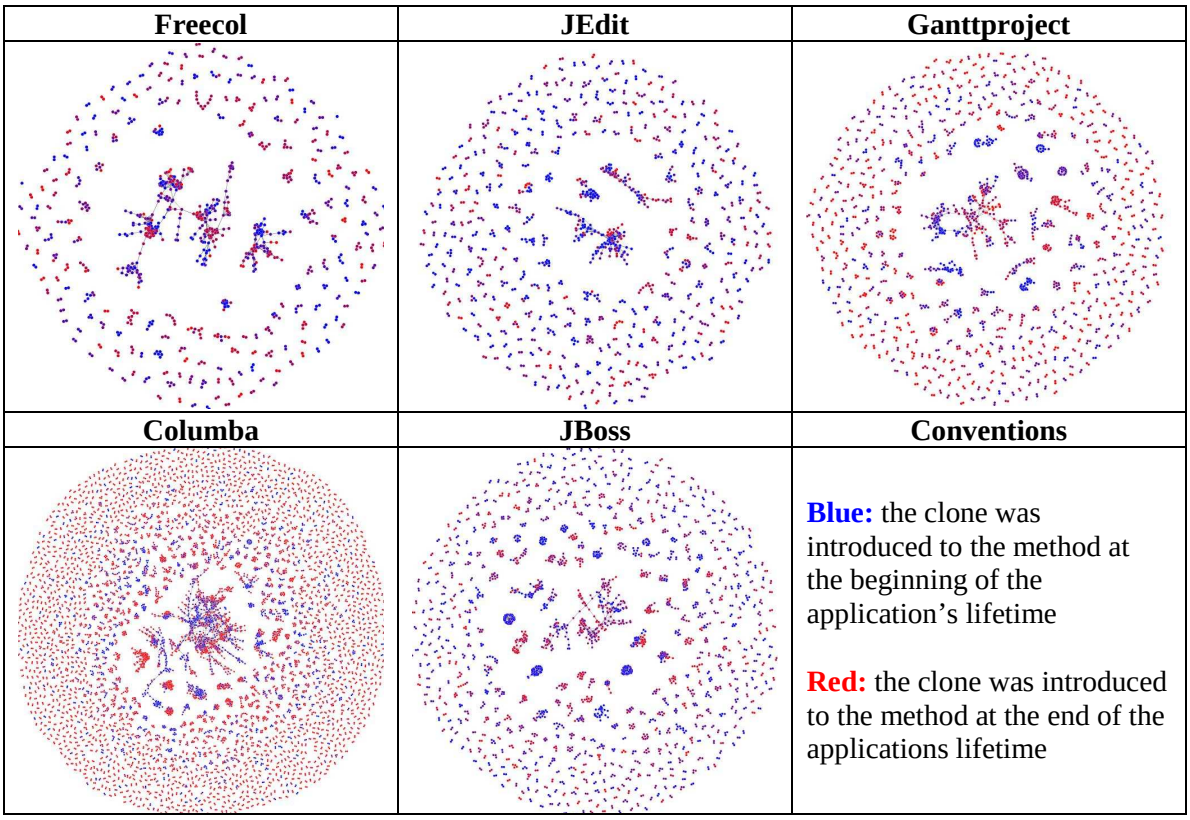


Figure 8-5. Commit creation of clones.

We have validated the intuition obtained from analyzing the Figure 8-5 by calculating the percentage of methods that become cloned every tenth of the history analyzed per application. We have found that indeed, there is no pattern between the time in the history of the application and the creation of clones in methods that were not cloned before, as Table 8-2 shows. While

the majority of methods become cloned at the beginning of the history analyzed on Freecol, JEdit, and JBoss; the majority of methods become cloned at the end of the history analyzed on GanttProject and Columba; the difference in the latter could be due to large restructuring commits performed in the last commits analyzed of these applications.

Table 8-2. Percentage of methods that become cloned per commit transaction intervals.

Commits	Freecol	JEdit	Ganttproject	Columba	JBoss
0% - 10%	26%	43%	15%	13%	30%
10% - 20%	7%	3%	5%	4%	7%
20% - 30%	13%	4%	2%	3%	6%
30% - 40%	8%	8%	10%	4%	8%
40% - 50%	7%	13%	11%	3%	11%
50% - 60%	5%	10%	3%	4%	11%
60% - 70%	14%	6%	15%	3%	8%
70% - 80%	8%	5%	6%	4%	11%
80% - 90%	6%	7%	8%	4%	3%
90% - 100%	8%	2%	26%	59%	5%

8.2.3 Eliminator

The **eliminator** characteristic refers to the developer that eliminates the last clone relation of a method. This characteristic shows to what extent clones are eliminated from methods, if the same developer eliminates all clones in a family or not, and if there is a particular set of developers responsible for the elimination of clones. Again, before concluding anything about the eliminators one should find out if they are the same that are usually in charge of the majority of the application.

Figure 8-6 shows all the methods that at some moment were free of cloned fragments after having been cloned. Each color identifies a different developer, and means that the developer is the responsible for eliminating the last cloned fragment in the method. We have found that in very few methods all their cloned fragments are eliminated. In fact, most of the clones are eliminated because one of the methods in the clone relation is deleted. There does not seem to be a single responsible for the elimination of clones, in fact the diversity of eliminators is much higher than the diversity of creators, probably all developers tend to eliminate clones as they find them. There is no pattern about the responsible for the elimination of cloned fragments of the same family: sometimes the cloned fragment is erased in only one of the methods that form the clone relation, which is enough to eliminate families of just two members (which are the

majority). However, there are cases in which all fragments of the family are eliminated. The number of cloned fragments eliminated in a family depends on the developer that eliminates the clone. See for instance that the dark-green and the gray developers in JEdit tend to erase the whole family, while the pink developer does not. JBoss presents the contrary pattern: most of the developers eliminate just one fragment, except for the yellow-orange developer that tends to eliminate the fragments of whole family.

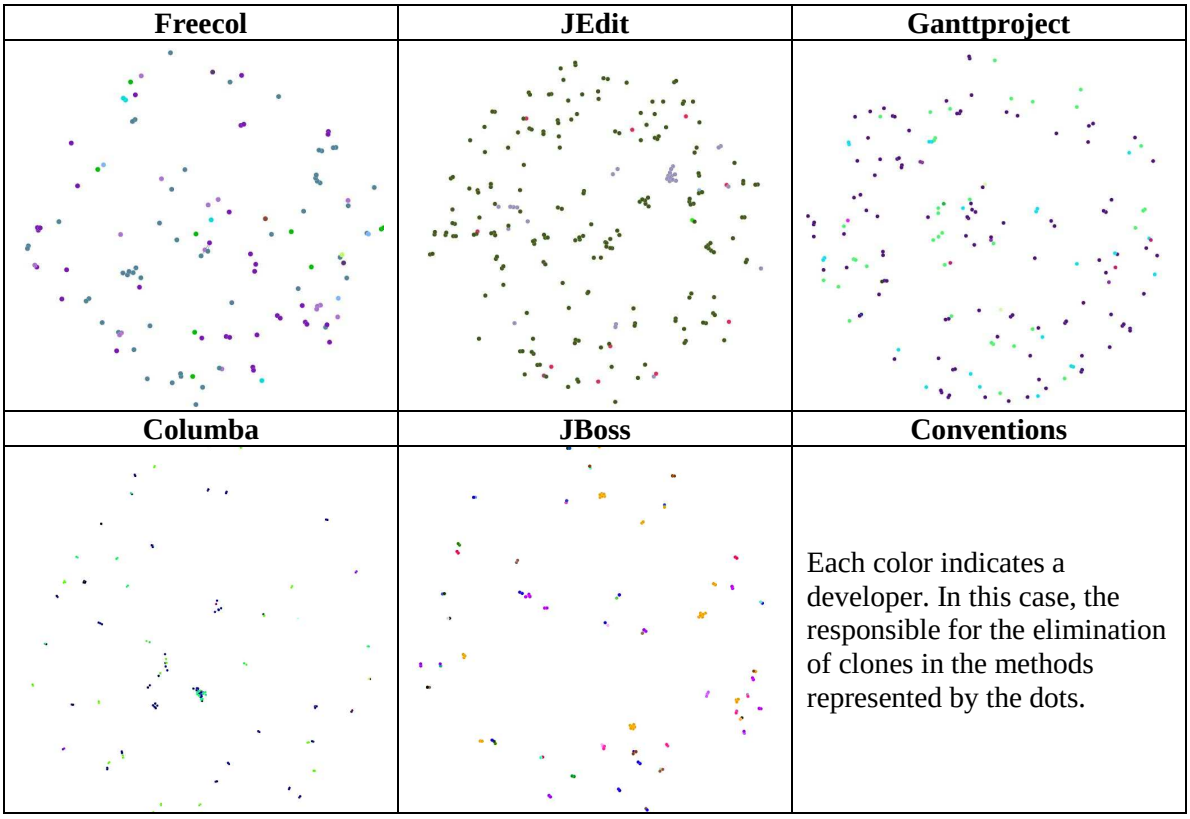


Figure 8-6. Eliminator of clones.

We have analyzed the intuition given by Figure 8-6 that all developers are evenly responsible in the elimination of clones by comparing the percentage of methods cloned and the percentage of clones eliminated for the developers that converted the majority of methods into cloned methods (see Table 8-3). Contrary to the visual intuition, we found that the developers that inject clones tend to be the ones that delete them.

Table 8-3. Developers that cloned the highest amount of methods (in decreasing order), compared with the percentage of clones deleted per developer.

Freecol		**JEdit**		**Ganttproject**		**Columba**		**JBoss**	
Total deleted: 12.8%		Total deleted: 19.3%		Total deleted: 9.4%		Total deleted: 4%		Total deleted: 9%	
added	deleted	added	deleted	added	deleted	added	deleted	added	deleted
69%	5%	95%	16%	56%	6%	92%	2%	17%	1.1%
17%	4%	3%	1%	18%	2%	3%	0.7%	16%	0.9%
6%	2%	1%	0%	9%	1%	2%	0.6%	15%	0.5%
3%	0.2%	1%	2%	6%	0.1%	1%	0.1%	12%	0.2%
2%	1%	0.4%	0.1%	3%	0%	1%	0.3%	8%	0.2%

8.2.4 Elimination

The **elimination** characteristic refers to the logical change/commit transaction in which the last cloned fragment of a method stop belonging to the clone family. This characteristic would show us if the fragments of a clone family are usually eliminated in a burst of close commits, or if they are eliminated in disperse commits.

Figure 8-7 shows that there is no identifiable pattern for the elimination of clones. Some of the clones are deleted at the beginning of the application's history (blue dots), while others are deleted at the end of the application's history (red dots). However, cloned fragments of the same family are eliminated in close commits; notice that dots in the same cluster have similar colors.

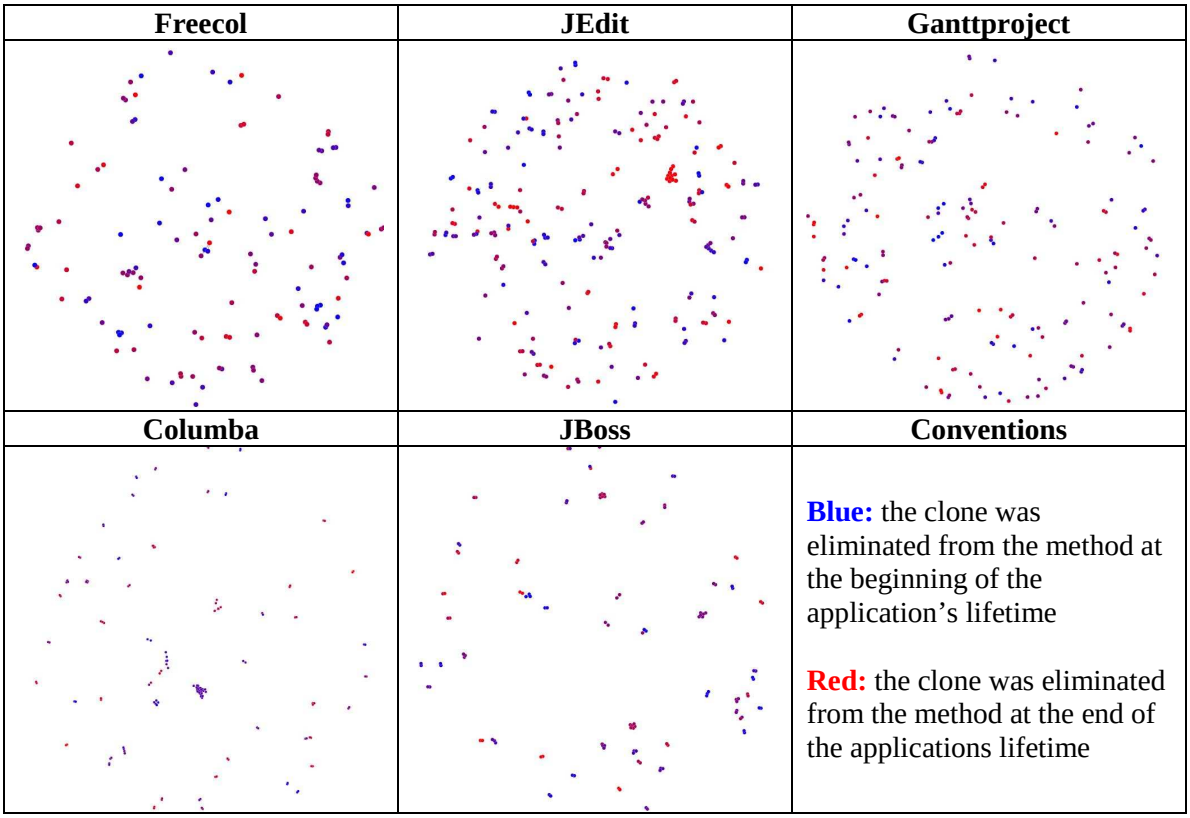


Figure 8-7. Commit elimination of clones.

We confirmed that there is no particular moment in which clones are deleted by counting the amount of methods that become clone-free on each tenth of the commit transactions of the applications, as Table 8-4 shows.

Table 8-4. Percentage of methods that become clone-free per commit transaction intervals.

Commits	Freecol	JEdit	Ganttproject	Columba	JBoss
	Total deleted: 12.8%	Total deleted: 19.3%	Total deleted: 9.4%	Total deleted: 4%	Total deleted: 9%
0 – 10%	2.7%	3.0%	0.4%	0.4%	1.5%
11 – 20 %	0.5%	1.9%	0.4%	0.3%	0.7%
21 – 30 %	0.5%	1.8%	0.9%	0.4%	0.5%
31 – 40 %	1.0%	0.8%	0.2%	0.9%	0.5%
41 – 50 %	1.5%	2.5%	0.8%	0.6%	2.5%
51 – 60%	1.8%	1.7%	0.6%	0.3%	0.9%
61 – 70 %	1.5%	1.5%	2.3%	0.2%	1.4%
71 – 80 %	0.7%	1.3%	1.3%	0.5%	0.8%
81 – 90 %	1.2%	2.4%	1.2%	0.2%	0.6%
91 – 100 %	1.4%	2.4%	1.2%	0.5%	0.6%

8.2.5 Method ownership vs. clone ownership

The **ownership** characteristic refers to the developer that changes the method most in an interval of time. The ownership of the method refers to the developer that changes the method the most along its whole lifetime. The ownership of the clone refers to the developer that changes the method while it is cloned. Comparing the ownership of the method along its lifetime vs. when it is cloned would indicate if the person responsible for changing the method is also the responsible for the changing the clone or not. Different owners for the method and for the clone would indicate different change patterns when methods are cloned, and possibly collective maintenance of clones as suggested in [Balint '06].

The methods without an owner (i.e. a developer that is responsible for the majority of its changes) are excluded from the diagrams. We have found that several methods do not have an owner, because they never change (see Figure 8-8). Whenever the method has an owner, she tends to be the owner of the clone, marked in the figures as green dots. That means that the person responsible for the method is usually the person that handles the clone. Many methods with owner, never change while they are cloned, those methods are marked with blue dots; which indicates stable clones. Finally, the minority of methods has a different owner for the method and for the clone; those methods are marked with red dots. That would mean that the minority of methods have a developer responsible for the method when cloned and another developer responsible for the method when it is not cloned. However, given that the ownership of the method includes the period cloned and the fact that the methods cloned remain most of their lifetime cloned, the results are biased towards having the same developer when cloned and when not cloned.

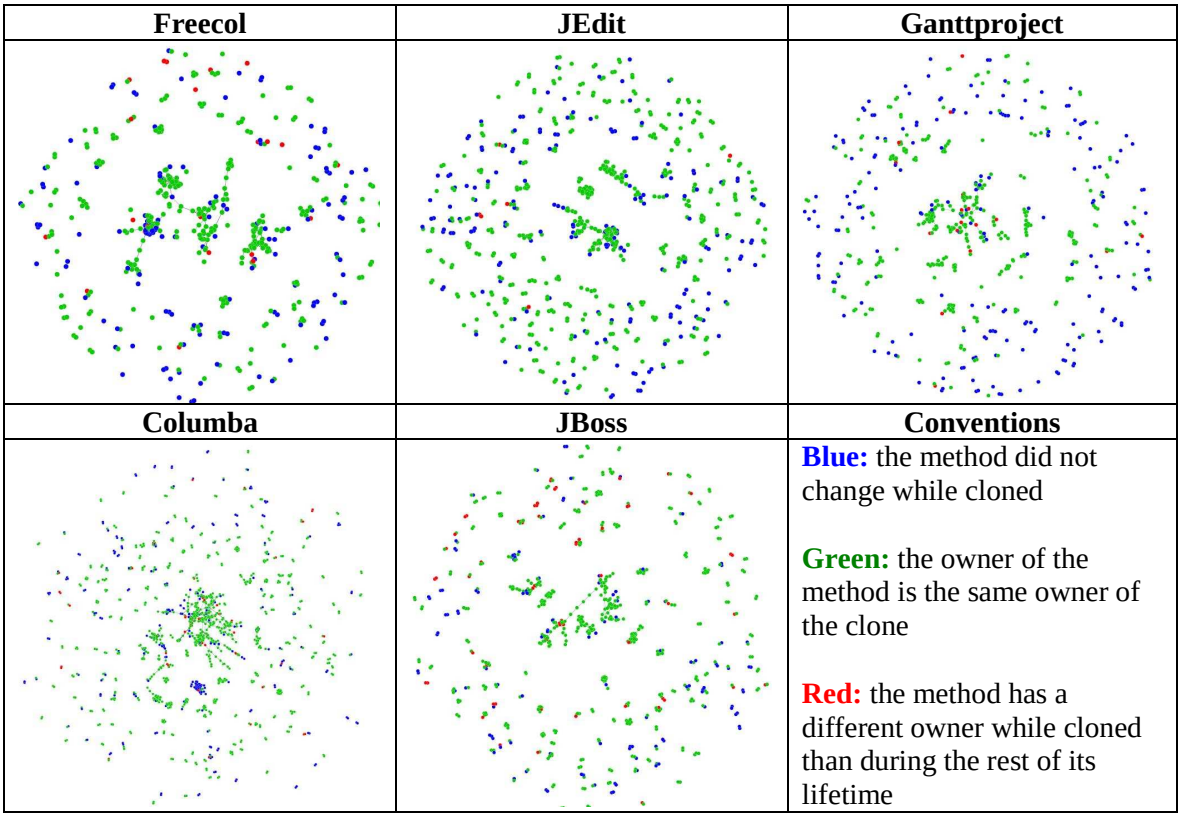


Figure 8-8. Ownership of methods vs. ownership of clones.

8.2.6 Size of cloned fragments

The characteristic **tokens cloned** refers to the number of tokens cloned inside a method. This characteristic allows knowing if clones are mostly small fragments code, and in that sense, they can be improved with syntactic sugar such as macros, or if they are mostly large fragments, and therefore they may indicate design issues. The size of the cloned fragments of a method is calculated as the average of the size of all cloned fragments inside that method, during their lifetime.

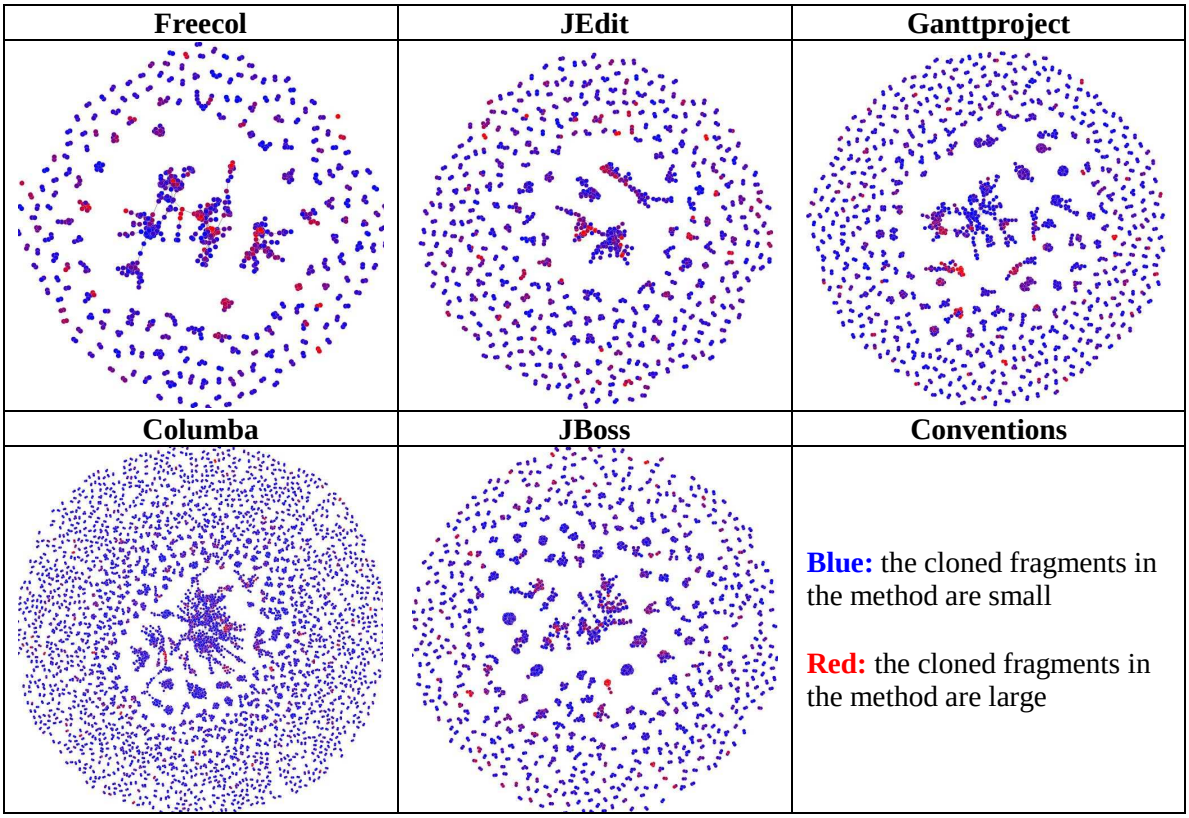


Figure 8-9. Tokens cloned in methods

Figure 8-9 shows the size of the cloned fragments on each method as a color scale that goes from blue for very small fragments to red for very large fragments. We have found that the majority of methods have small clone fragments, i.e. between 30 to 40 tokens. Several families have large fragments, i.e. between 100 and 200 tokens. The minority of families have very large cloned fragments i.e. more than 200 tokens. We expected large families to be related by small fragments (i.e. large blue clusters); however, this cannot be confirmed because although there are large families cloned by small fragments, there are also large families cloned by large fragments (i.e. large red clusters).

We have tested the hypothesis provided from the analysis of Figure 8-9 by finding the percentages of methods that have certain amount of tokens cloned, as Table 8-5 shows. We have found that the majority of methods analyzed (over 56%) have at most 50 tokens cloned, and that methods with more than 100 tokens cloned are rare (at most 15%).

Table 8-5. Percentage of methods per intervals of number of tokens cloned.

Size interval	Freecol	JEdit	Ganttproj.	Columba	JBoss
0 – 50 tokens	68%	76%	63%	56%	66%
50 – 100 tokens	21%	17%	27%	29%	22%
100 – 150 tokens	6%	4%	4%	8%	7%
150 – 200 tokens	2%	1%	3%	3%	2%
More than 200 tok.	3%	2%	3%	4%	3%

8.2.7 Percentage of the method affected by cloning

The characteristic **percentage cloned** refers to the percentage of tokens in the method that are cloned. The percentage affected would allow us to know if cloned fragments occupy most of the method or not. It is important to measure not only the size of the cloned fragments, but also to what extent they affect the method because it could be that most of the fragments are very small, but they compose most of the method.

Figure 8-10 indicates the percentage of cloning in a method the whole period it is cloned. If the dot representing the method is blue it means that the percentage of the method that has tokens cloned is very low, conversely if the percentage of the method tokens is close to 100% the method would be red. We have found that methods tend to be highly cloned whenever they have several cloned fragments.

The analysis of Figure 8-10 indicates that most of the tokens in cloned methods are part of cloned fragments; we have tested such hypothesis by finding the percentage of methods that have certain percentage of tokens cloned, as Table 8-6 shows. We have found that the majority of methods analyzed have a very low or a very high percentage of tokens cloned; note that the first and last intervals (0-10% and 91%-100%) contain the highest figures for each application analyzed. There are applications like Freecol and JEdit that have an even distribution of the percentage cloned in their methods. However, Ganttproject, Columba, and JBoss show a tendency for methods highly affected by clones as the number of methods that have at least 70% of their tokens cloned is above 50%.

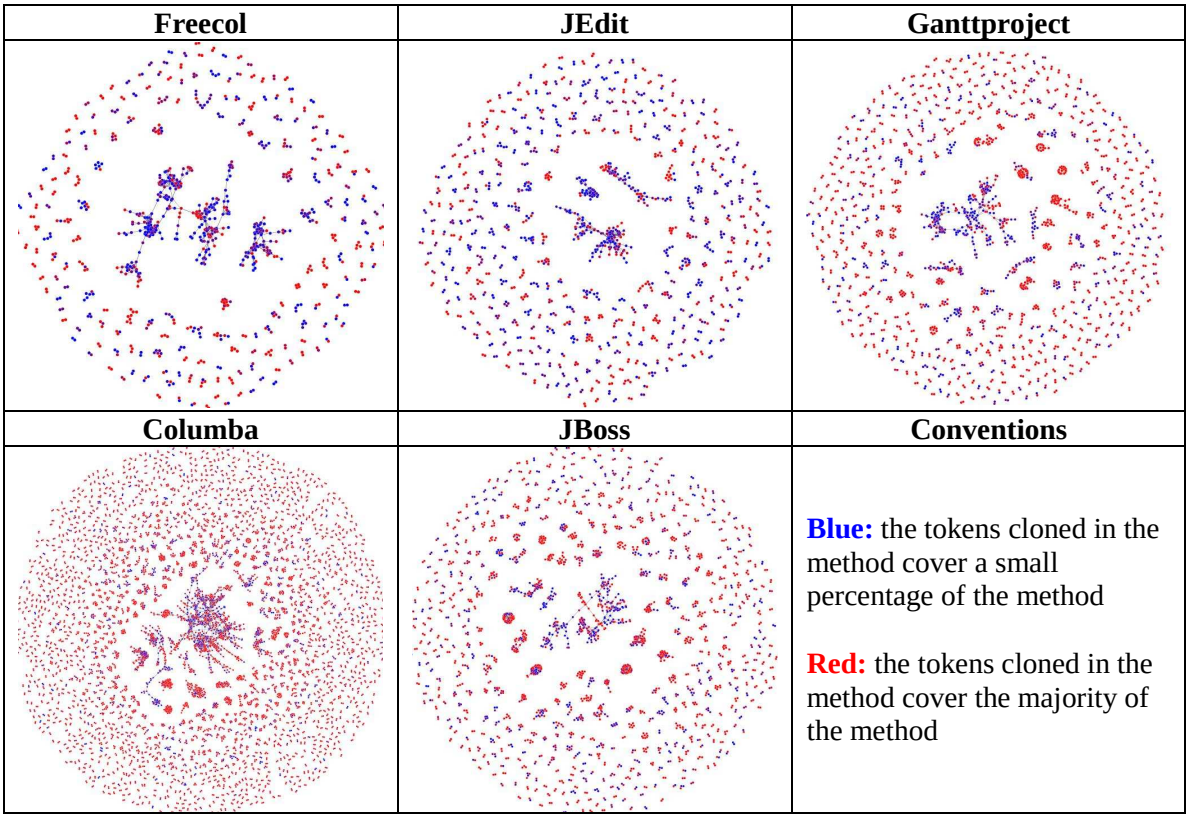


Figure 8-10. Percentage of the method cloned

Mixed with the previous results, it means that although most of the clones are small they cover most of the methods they affect, so the methods affected by cloned tend to be small. In fact, the tendency of JEdit and Freecol of having more methods with a low percentage cloned than the rest of the applications might be because the methods in these applications tend to be larger (19 LOC for both JEdit and Freecol in average) than the methods in the rest of applications (14 LOC for Ganttproject, 15 LOC for JBoss, and 16 LOC for Columba).

Table 8-6. Percentage of methods per intervals of percentage of tokens cloned.

Percentage cloned interval	Freecol	JEdit	Ganttproj.	Columba	JBoss
0 – 10%	22%	24%	12%	4%	12%
11 – 20 %	9%	9%	4%	3%	5%
21 – 30 %	8%	8%	5%	3%	5%
31 – 40 %	6%	8%	4%	2%	4%
41 – 50 %	5%	6%	4%	3%	5%
51 – 60%	4%	4%	4%	2%	4%
61 – 70 %	5%	3%	3%	2%	4%
71 – 80 %	5%	4%	4%	12%	5%
81 – 90 %	4%	3%	4%	20%	5%
91 – 100 %	27%	24%	50%	44%	41%
less than 30%	39%	41%	21%	10%	22%
more than 70%	36%	31%	58%	76%	51%

8.2.8 Percentage of the method's lifetime affected by cloning

The characteristic **lifetime cloned** refers to the percentage of commit transactions of the method's lifetime in which the method has cloned fragments. The percentage affected would allow us to know if cloned methods have clones most of their lifetime or not. Although Kim et al. [Kim '05] have found that clones are usually modified early on and their similarity vanishes, we do not know yet if there are methods more susceptible to keep clones along their life or not.

The clone graphs in Figure 8-11 represent the percentage of time that the method has been cloned, from the time it has been part of the application as a scale of colors going from blue to red. Notice that, in Figure 8-11, that there are several families with diverse lifetime cloned. This is because the lifetime percentage cloned is dependent of the length of the lifetime of each method, so although the family lasts 100 commit transactions, the lifetime percentage would be different if one of the methods in the family was alive for 1000 commits (10%) and another one for just 150 commits (66%).

The purpose of this analysis is to compare the lifetime in which a method is cloned with the lifetime of clone fragments. Given the results presented by Kim et al. showing that cloned fragments tend to be deleted rapidly after inserted [Kim '05] , we expected blue to be the dominant color in the figures. However, it seems that many cloned methods are cloned most of their lifetime, especially those involved in large and complex families. We confirmed this graphical observation by finding the percentage of methods that have certain percentage of

lifetime cloned, which is shown in

Table 8-7.

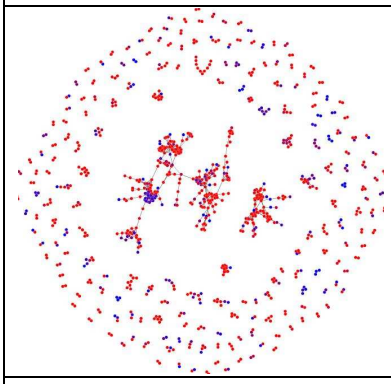
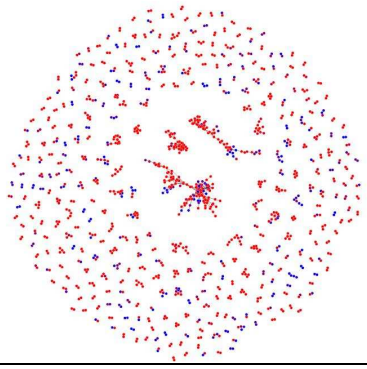
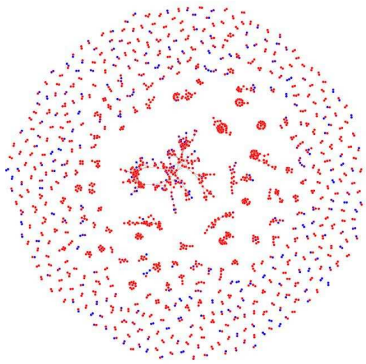
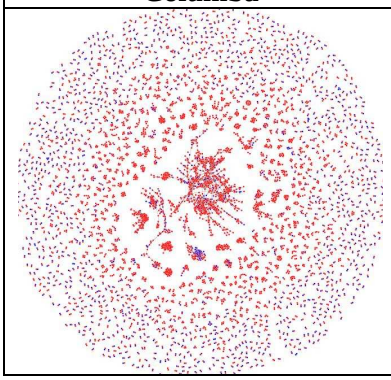
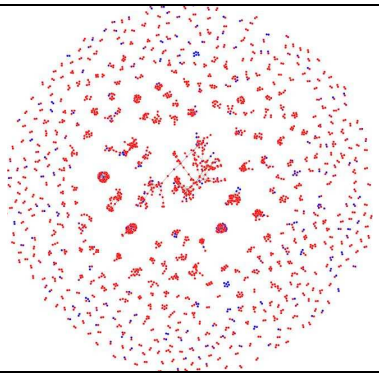
Freecol	JEdit	Ganttproject
		
Columba	**JBoss**	**Conventions**
		**Blue:** the minority of the time that the method has been part of the application has been cloned **Red:** most of the time that the method has been part of the application it has been cloned

Figure 8-11. Lifetime of the method cloned

Table 8-7. Percentage of methods per lifetime cloned.

Lifetime cloned interval	Freecol	JEdit	Ganttproj.	Columba	JBoss
0 – 10%	6%	9%	8%	20%	7%
11 – 20 %	3%	4%	2%	1%	2%
21 – 30 %	5%	4%	3%	2%	2%
31 – 40 %	4%	3%	2%	1%	2%
41 – 50 %	4%	3%	1%	1%	3%
51 – 60%	2%	3%	1%	1%	1%
61 – 70 %	3%	3%	2%	1%	2%
71 – 80 %	3%	2%	2%	1%	2%
81 – 90 %	4%	3%	2%	1%	3%
91 – 100 %	66%	67%	76%	71%	77%
less than 30%	14%	17%	13%	23%	11%
more than 70%	73%	72%	80%	73%	82%

This result apparently contradicts Kim et al.'s result. A possible explanation for the contradiction is that the lifetime of cloned methods is very volatile. However, we found that the lifetime of cloned methods is larger than the lifetime of methods not cloned (see Table 8-8).

Table 8-8. Average lifetime of methods not cloned vs. cloned methods

	Avg. lifetime of methods not cloned	Avg. lifetime of cloned methods
Freecol	466 commits	536 commits
JEdit	519 commits	607 commits
Ganttproject	270 commits	493 commits
Columba	580 commits	590 commits
JBoss	787 commits	863 commits

Another possible explanation is that although clone fragment and their families are volatile, the cloned relations are persistent. In other words, cloned fragments are volatile because they are changed in such a way that they cannot be recognized anymore as part of that family anymore. However, the fragment is still a cloned with new family formed of parts that are still common among the methods that used to form the family of the cloned fragment, i.e. still cloned to many of the other cloned fragments in the previous family but with a different common part. Therefore, cloned fragments just change of family (or common clone) as they evolve.

For instance, suppose that the method `m1` Figure 5-5 evolves as Figure 8-12 shows. Note that although only the cloned fragment of method `m1` changed, the clone-family disappears because the clone (see Figure 5-6) shared by the two methods changes. Notice also that although the

cloned fragment and the cloned family disappeared, the cloned relation did not; with the changes, there is a newer cloned fragment and a new clone family among the same set of methods. This is a very common situation when clones in methods change.

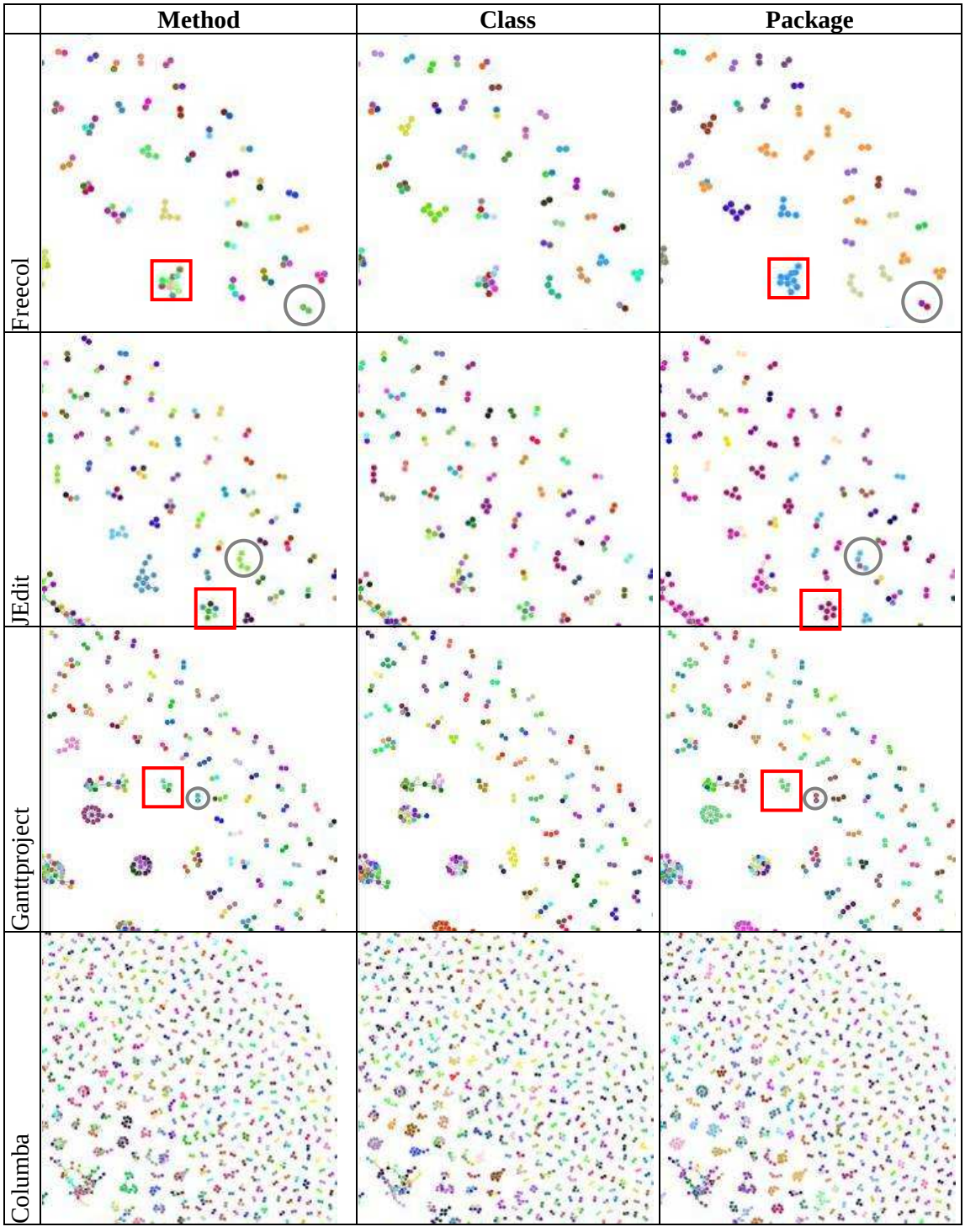
```
public int m1(int a, int b
, int c){
   if (a % 2 == 0){
      return a + b + c;
   }
   b *= 2;
   return b + c + c * 2;
}
```

```
public int m2(int x, int y
, int z){
   if (x % 2 == 0){
      return x + y + z;
   }
   z *= 2;
   return y + z + x * 2;
}
```

Figure 8-12. Example of evolution of the clone relation on Figure 5-5

8.2.9 Full method name

The **name** characteristic refers to the full name of the method that has the fragment cloned. According to the findings in [Golomingi '01; Kapser '03; Kapser '04; Jarzabek '06] cloned fragments of the same family usually occur in the areas that are close to each other, either in terms of directories [Kapser '03; Kapser '04] or in terms of class hierarchies [Golomingi '01; Jarzabek '06]. We think that the reason for these findings is that clones are usually located in SCEs that have similar responsibilities, and that the name of the SCEs reveals such similarities in the responsibilities. In fact, some authors have used the name of the methods to find clones [Marcus '01; Calefato '04], and to eliminate false positives [Burd '02]. Therefore, if cloned fragments are in methods with a very similar name, they are likely to be in different files that perform parallel functionalities, e.g., sibling classes. If cloned fragments of the same family are in different files, they are likely to be in the same package or in a package with a similar functionality. However, if the methods have a different name, it is likely that they are located in the same file.


Method
Class
Package
Freecol
JEdit
Ganttproject
Columba

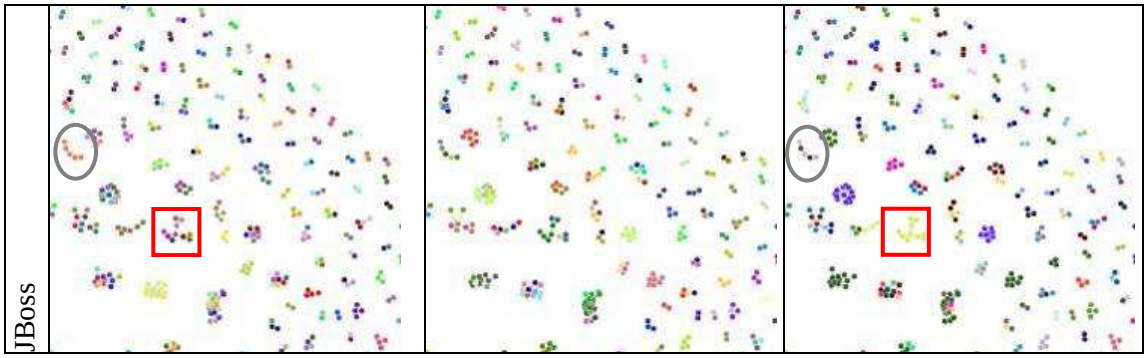


Figure 8-13. Similarity of the names of the methods, classes and packages of cloned fragments of the same family

Figure 8-13 has a different color for each method name, package name, and class name of the application. For instance, to obtain the graphs of the first column, the application orders lexically the names of all the methods that were cloned. Afterwards, it assigns a different consecutive color to each method name, in that way methods with a similar name would have a similar color. Similarly, the second column assigns colors depending only on the name of the classes of the methods with clones, and the third column assigns colors depending only on the name of the packages of the methods with clones.

This means that it is possible to spot if the names of the methods of the same family have a similar method name, because they would look like clusters with a similar color. Given that there is a large variety of colors, and that they are difficult to distinguish in the reduced figure, the figures only show the top right corner of every application. Results suggest that the similarity among the names of the methods of the same family is not as large as other researchers supposed (see first column in Figure 8-13). Most of the clones of the same family are located in the same package (see second column in Figure 8-13). We have also found that although not always the names of the methods of a family are similar, the names of their classes or packages are similar. Most of the clone families whose methods have similar names are in different classes of the same package. Clone families whose methods are in different packages tend to have a similar name (gray circle). Clone families whose methods are in the same package tend to have different names (red square). Note also, that several packages are involved in cloning, without any predominant package.

8.2.10 Role

The **role** characteristic refers to the way in which the first cloned fragment of a method is

created. There are three possible roles: seed, copy, and twins. The idea is to analyze the clone relation that is associated with the creation of the cloned fragment. If the cloned fragment is not changed, the clone relation was created because the other method in copied the clone from the cloned fragment in the method analyzed. In this case, the cloned fragment is the **seed** of the clone. If the cloned fragment analyzed changes but the other cloned fragment that forms the clone relation does not change, then the cloned fragment analyzed is the **copy**. Finally, if both cloned fragments are changed when the clone relation is created, is because the cloned fragment was introduced to both methods at the same time, in this case both cloned fragments are identified as **twins**. We are the first ones to propose this classification, so the main goal is to find out how many of the clones are created by copy and paste, and how many are created because the developer realized that a set of methods needed similar functionality.

The clone graphs in Figure 8-14 depict copy clones in red, twins in yellow, and seeds in green. We have found that for most of the applications, the majority of the clones are twins (yellow), which means that most of the cloned fragments are introduced at the same time in all methods of the family. The only application that does not have as majority the twins role is Columba, whose majority of clones are seeds (green), which is the second most common category for the rest of applications. Note that several families have all their members as seeds. These families introduced their cloned fragments at the same time. They are identified as seeds because they are cloned at the beginning of their lifetime; but the creation of the methods is not modeled as a change. Finally, it is remarkable that the small minority of cloned fragments are identified as copy (red dots), which is contrary to intuition.

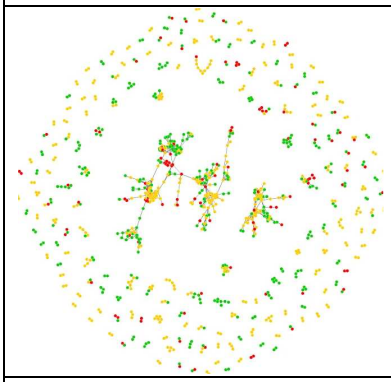
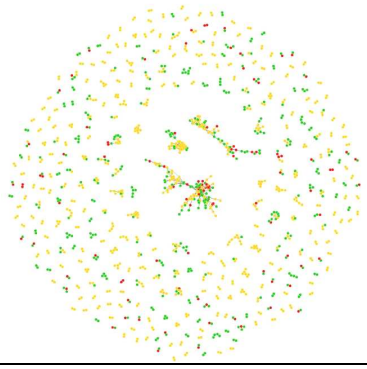
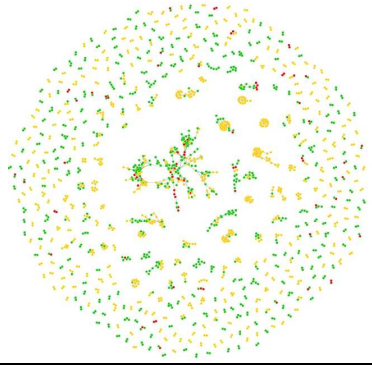
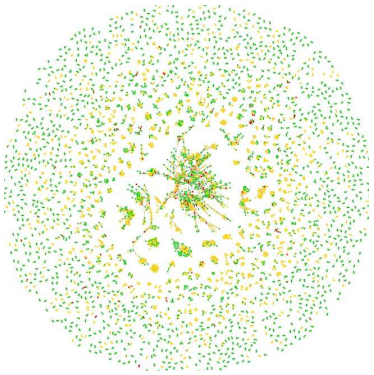
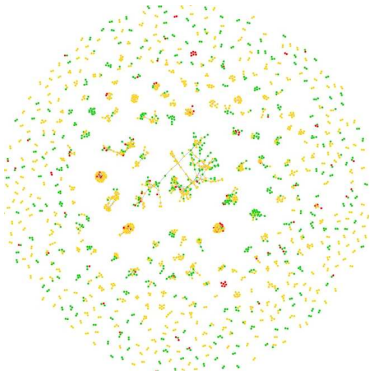
Freecol	JEdit	Ganttproject
		
Columba	**JBoss**	**Conventions**
		**Green:** Seed. Other method copied a fragment of this method. Or the methods were created cloned **Yellow:** Twin. The cloned fragment was pasted in methods of the same family. **Red:** Copy. The cloned fragment was pasted in this method

Figure 8-14. Creation role of clones

In order to assess the accuracy of the visual inspection we counted the percentage of methods per role, and reported on Table 8-9. Table 8-9 shows that the visual inspection was accurate in identifying the frequency of roles. Therefore, the visual inspection proposed is a reliable way to detect patterns when there are few and very different colors.

Table 8-9. Percentage of methods per role.

Role	Freecol	JEdit	Ganttproj.	Columba	JBoss
Seed	39%	32%	43%	60%	37%
Twins	49%	60%	52%	38%	57%
Copy	12%	8%	4%	2%	5%

8.2.11 Age of the method when cloned

The **age** characteristic refers to the commit transaction in which the first cloned fragment is added to the method. The age is relative because it depends on the commit transaction in which

the method is created, i.e. it says the number of commit transactions before the method is cloned. Monden et al. found that younger code is the most likely it is to be cloned; however, we would like to know how fast methods acquire clones.

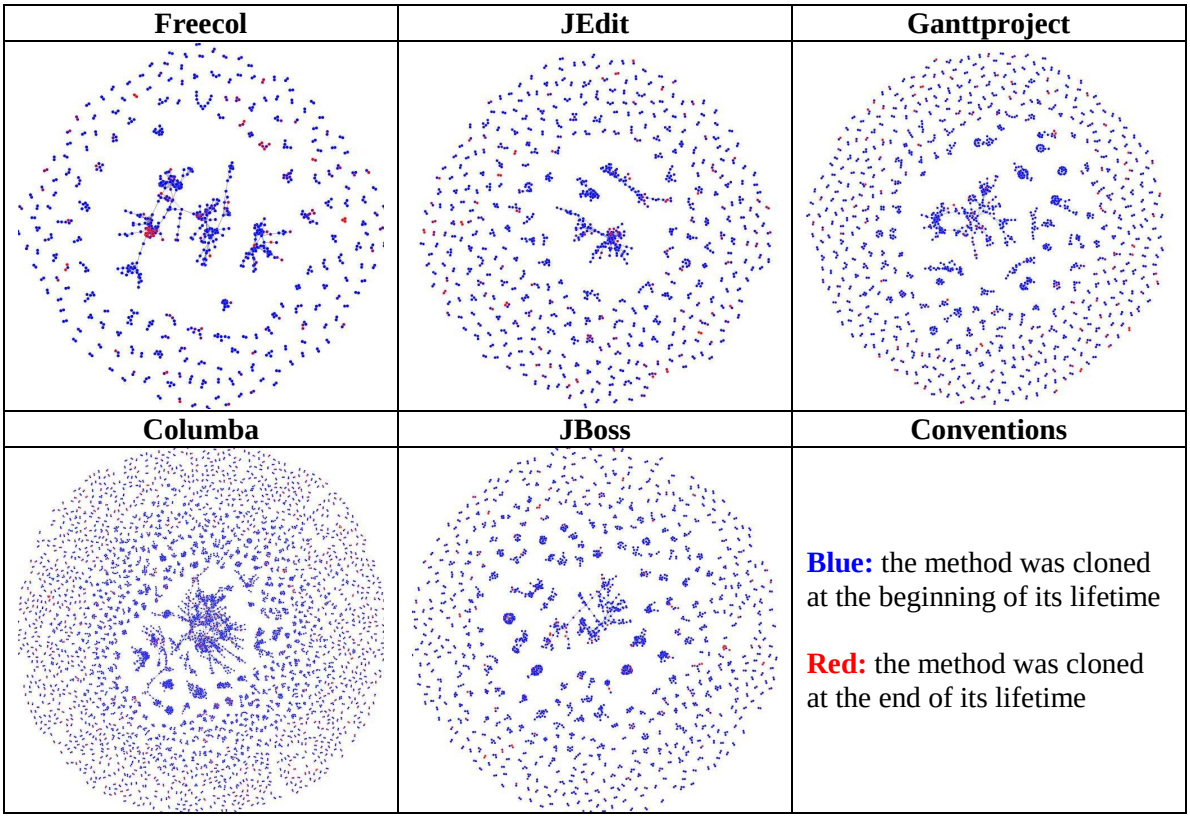


Figure 8-15. Age of the method when the first cloned fragment is introduced

As other characteristics regarding scales, we have depicted the age of the method when is cloned by first time as a color scale. The color scale goes from blue when it is early in the method lifetime to red when it is close to the elimination of the method. As Figure 8-15 suggest, most of the clones are introduced close to the creation of the method, which is consistent with the great amount of 'seed' and 'twins' roles found.

Table 8-10. Percentage of methods per number of commits before becoming cloned.

Commit transactions before becoming cloned	Freecol	JEdit	Ganttproj.	Columba	JBoss
0 or 1	68%	76%	80%	74%	80%
Less than 50	7%	6%	4%	3%	5%
Less than 100	4%	2%	2%	1%	2%
Less than 150	3%	1%	1%	2%	1%
Less than 200	2%	1%	1%	1%	1%

In order to assess the accuracy of the visual inspection we counted the percentage of methods per number of commits before their first clone, and reported it on Table 8-10. Table 8-10 validates the visual inspection showing that between 68% and 80% of the methods are created cloned.

8.2.12 Dissimilarity

The **dissimilarity** characteristic refers to the percentage of tokens that are different in the cloned fragments of a method with respect to their families. The dissimilarity of a method is the average dissimilarity of its cloned fragments along their lifetime. It seems that the level of dissimilarity is directly related with the amount of cloned fragments (see section 3.3.3) in a clone family.

The percentage of tokens that differ among the cloned fragments of a family is depicted as a color scale. The highest level of differences among the cloned fragments in a family is 45% because that is the maximum percentage of differences allowed by the false positive filter (as explained in section 5.2.1). Low percentages of differences are depicted in blue-like colors, while high percentages of differences are depicted in red-like colors. Whenever the cloned fragments in the family are exact, the methods are depicted in black. Given that a method can have several cloned fragments, and given that the level of differences can change over time; we have decided that the value for each method is the average for all its cloned fragments across its lifetime.

Results in Figure 8-16 suggest that cloned fragments have very few differences. In general, less than 10% of the tokens cloned are different across fragments of the same family. We could not verify that the amount of differences is related to the size of the clone family, note that there are several small families that tend to be red, as well as, several large families that tend to be blue.

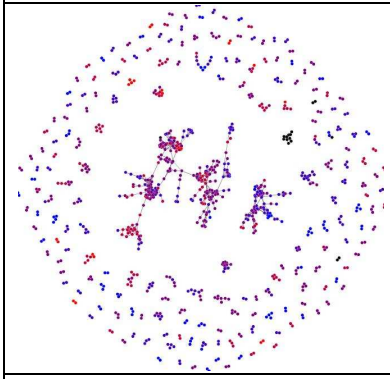
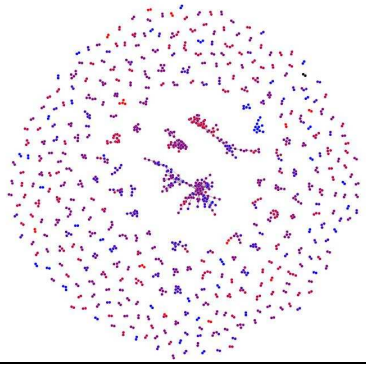
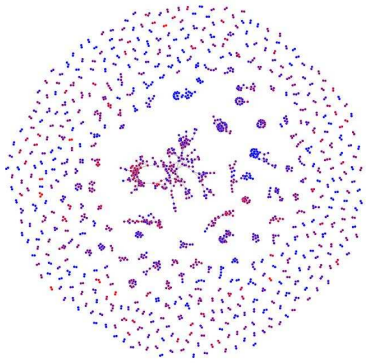
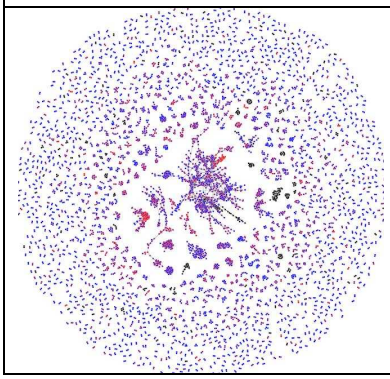
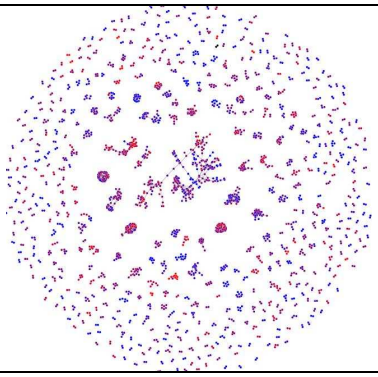
Freecol	JEdit	Ganttproject
		
Columba	**JBoss**	**Conventions**
		**Blue:** the percentage of tokens that are different among the fragments of the same family tends to zero. **Red:** the percentage of tokens that are different among the fragments of the same family tends to 44%.

Figure 8-16. Percentage of differences on cloned fragments of the same family

In order to assess the hypothesis that cloned fragments do not differ much from the other clones of their family we counted the percentage of methods per percentage of different tokens in the clone that identifies their family (see Figure 5-6), and reported it on Table 8-11. Table 8-11 contradicts the visual intuition, showing that, for most of the applications, the percentage of different tokens in a clone is between 21% and 30%. This inaccuracy of the visual analysis could be because there is not enough difference between the purple and the blue colors that identify the percentages between 21% to 30%, and between 0 to 10%. Besides, the fact that Columba, which has largest number of methods cloned, is the one that has the highest percentage of methods with cloned fragments that differ from their families in less than 10% of their tokens may also have affected the conclusions.

Table 8-11. Percentage of methods per intervals of percentage of tokens different.

Percentage of tokens different	Freecol	JEdit	Ganttproj	Columba	JBoss
Exact clones	2%	0%	0%	5%	0%
0 – 10%	11%	8%	24%	40%	19%
11 – 20 %	26%	19%	30%	27%	23%
21 – 30 %	41%	45%	35%	21%	35%
31 – 40 %	18%	26%	10%	6%	20%
41 – 45 %	2%	2%	1%	1%	2%

8.2.12.1 Percentage of literals

The **percentage of literals** characteristic refers to the percentage of tokens that are literals in the cloned fragments of a method (as explained in section 5.2.1). The percentage of literals [Walenstein '04; Kapser '07] has been suggested as a way to locate false positives because many literals may indicate that the cloned fragments in a family do not share the same semantics. We would like to see if it is common to have high level of literals, and if there is any characteristic that helps to locate them.

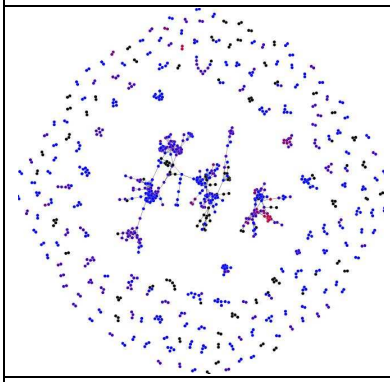
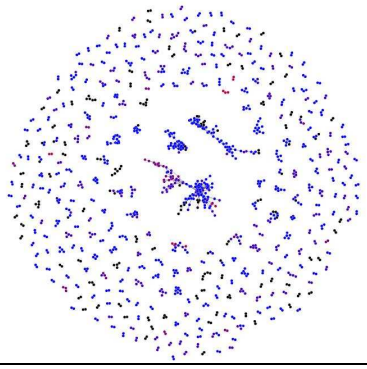
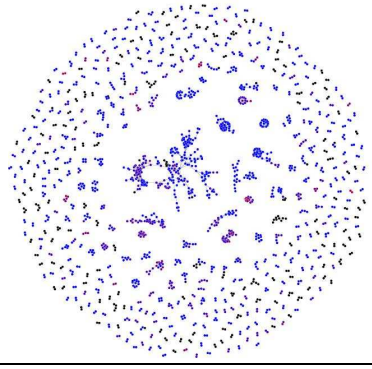
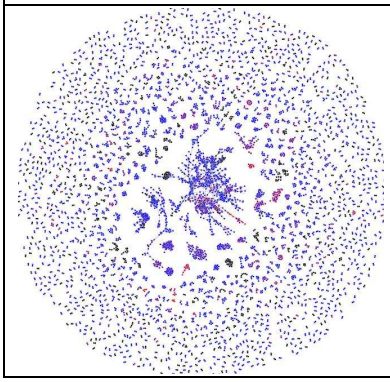
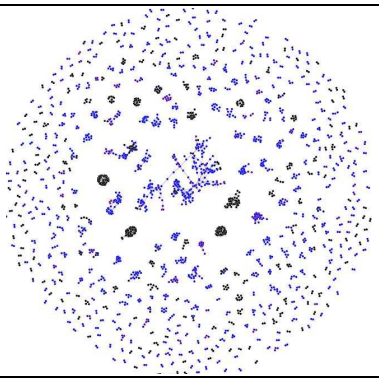
Freecol	JEdit	Ganttproject
		
Columba	**JBoss**	**Conventions**
		**Blue:** the percentage of literals in the clone family tends to zero. **Red:** the percentage of literals in the clone family tends to 44%. **Black:** there are no literals in the cloned fragment that represents the clone family.

Figure 8-17. Percentage of differences in literals on cloned fragments of the same family

The color scheme and the top threshold of the percentage of literals is the same that the one used for the percentage of differences. The percentage of literals indicates to what extent fragments are identified as cloned just because of large numbers of literals. Results coincide with the intuition that clones have a low percentage of literals given that their similarity is more likely to be due to syntactic similarities than to literals. Notice that several families (in JBoss in particular) that do not have any literals in their cloned fragments (black clusters). There are some families with high percentage of literals, but they tend to be small families.

Table 8-12. Percentage of methods per intervals of percentage of literals.

Percentage of literals	Freecol	JEdit	Ganttproj	Columba	JBoss
No literals in the clone	18%	17%	19%	19%	33%
0 – 10%	49%	56%	60%	72%	80%
11 – 20 %	29%	23%	18%	20%	19%
21 – 30 %	3%	3%	3%	6%	1%
31 – 40 %	1%	1%	1%	2%	0%
41 – 45 %	0%	0%	0%	0%	0%

In order to check the accuracy of the visual inspection we counted the percentage of literals in clones. The results are reported on Table 8-12, which shows that the visual inspection was correct in indicating that most of the clones have a low percentage of literals indicating that they are not false positives.

8.2.12.2 Differences in method-calls and types

The **difference in method-calls and types** characteristic refers to the percentage of tokens that refer to method-calls and types that are different among cloned fragments of the same family from the number of tokens that refer to method-calls and types in the clone that represents the family (as explained in section 5.2.1). The differences in method-calls and types has also been suggested as a way to locate false positives because indicate that cloned fragments in a family may not share the same semantics [Walenstein '04; Kapser '07]. We would like to see if it is common to differ in method calls and types, and if there is any characteristic that helps to locate clone families with high differences on method calls and types.

The level of differences in methods called or types referred is also shown as a color scale varying from black to blue to red when the percentage of differences is close to 45%. The results are similar to those found for the percentage of literals; that is, most of the cloned fragments have a very low percentage of differences in the methods called and types referred (Figure 8-18). Whenever there are differences, the level of differences tends to vary across the members of a family. This could indicate that they are unlikely to be accidental clones and they might be copy and paste clones due to an inappropriate design. Whenever the level of differences is high for all the members of the family (there is only a couple of cases in JEdit, Ganttproject, and JBoss), it might be due to accidental clones.

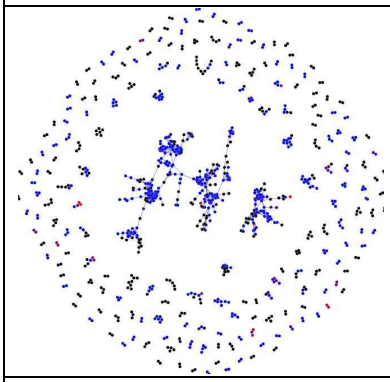
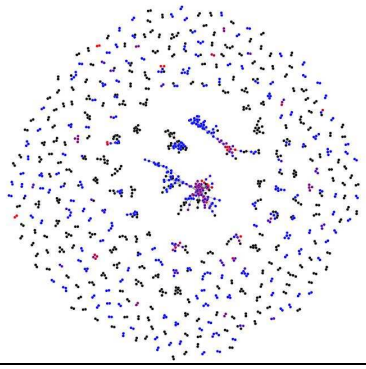
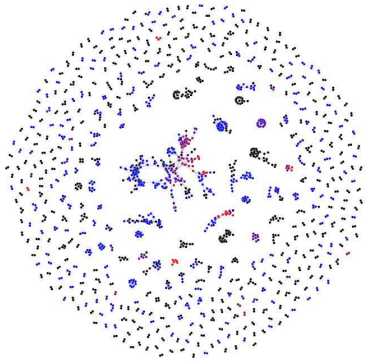
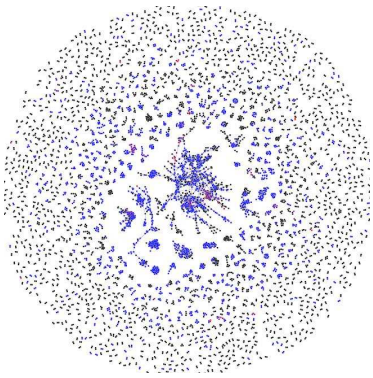
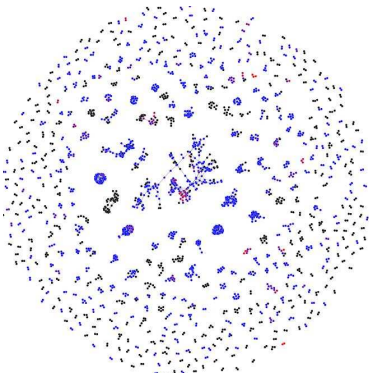
Freecol	JEdit	Ganttproject
		
Columba	**JBoss**	**Conventions**
		**Blue:** the percentage of method call or type tokens that are different among the fragments of the same family tends to zero. **Red:** the percentage of method call or type tokens that are different among the fragments of the same family tends to 44%. **Black:** there are no method call or type tokens in the family.

Figure 8-18. Percentage of differences in methods called and types referred on cloned fragments of the same family

In order to verify the accuracy of the visual inspection we counted the percentage of methods per percentage of method/type tokens that are different. The results, shown in Table 8-13, indicate that indeed the majority of clones either do not have any tokens that refer to types or to method calls (19% to 55%) or the percentage of tokens referring types or methods that are different is below 10% (35% to 58%).

Table 8-13. Percentage of methods per intervals of percentage of tokens of methods or types that are different.

Percentage of method/type tokens that are different	Freecol	JEdit	Ganttproj	Columba	JBoss
No method/type tokens	42%	52%	55%	19%	43%
0 – 10%	53%	36%	35%	58%	49%
11 – 20 %	3%	6%	5%	16%	6%
21 – 30 %	1%	4%	2%	5%	1%
31 – 40 %	1%	1%	2%	1%	1%
41 – 45 %	0%	1%	0%	0%	0%

8.3 Summary

This chapter has shown how to identify typical characteristics of the SCIUS. We have used GUESS to depict clone relations, and characteristics of clones. However, it is not the first time that GUESS has been used to analyze clones. In [Adar '07], the authors analyze the evolution of clone families using their graph representation to depict clone families, and their own scripts.

By analyzing typical characteristics in clones, we have found that:

- The main developer of the application is responsible for most of the clones, but the number of developers responsible for cloning increases as the contributions of the developers is more even. Then, clones seem to be a practice used by most of the developers.
- Most of the clone families are pairs of methods that are cloned at the same time, several of them since the creation of the method. Very few methods are cloned by copying and pasting from previous functionality, but rather by replicating new functionality. Given that clone families are usually composed of two methods, it is enough to change one of the cloned fragments to eliminate the clone family. In fact, the most of the clone relations are eliminated by eliminating the cloned fragment on one of the cloned methods, or by deleting one of the cloned methods rather than by restructuring the clone.
- Most of the clones disappear because the method that hosts them is deleted. Few developers eliminate cloned fragments from the methods, and the eliminator is usually a different developer than the creator of the clone.
- Clones and methods usually have the same owner and that in very few cases the owner of the method is different of the owner of the clone. However, a significant amount of methods does not have an owner because they did not change along their lifetimes.

- Most of the cloned fragments are small; these fragments tend to cover as much as nine tenths of the method. Whenever, the cloned fragment does not cover the majority of the method, the cover as little as one tenth of the method. Given that, most of the methods are highly cloned and that most of the cloned fragments are small, the evidence indicates that cloned methods tend to be small. Indeed, when looking at the data we found that methods with clones have in average 6 to 15 Lines Of Code depending on the application analyzed, while methods without clones have in average 25 to 33 Lines Of Code depending on the application analyzed.
- Methods that have cloned fragments are usually cloned all their lifetime. This means that cloned families are volatile, but clones in methods are not. Cloned fragments change with the same set of methods with which it is cloned. Therefore, although the cloned fragment changes, the cloned functionality remains most of the method's lifetime.
- Most of the clone families are within the same package. There are several similarities among the parts of the fully qualified name of the methods of a clone family. However, the similarities are too diverse to find any consistent pattern across the majority of clones.
- Finally, clones differ from between 20% to 35% of the cloned fragment; however, these differences are not in the tokens that call methods or types. In addition, the majority of the clones have a low percentage of literals.

The approach proposed by the methodology permits to discover a large amount of information about the SCIUS. However, a more systematic approach could have been used to explore composition of characteristics in interesting cases. For instance, to verify if clone families with high differences in the methods called were connected by large cloned fragments. This was implemented by depicting two characteristics at the same time; one characteristic is shown with the color, and the other the size of the circles representing the methods. However, the results were disappointing because the characteristic analyzed by size would make invisible many of the methods. Because those methods with a large value in the characteristic represented with the size covered the methods that are in their vicinity but have a small value in the characteristic represented by the size.

The following chapter presents how to analyze the evolution of SCIUS in SCEs, and its application on clones inside methods.

Chapter 9. Evolution of the Source Code Issue Under Study

This phase is composed of three sub-phases (see Figure 9-1): the evolution of the extension of the SCIUS in the application, the evolution of the persistence of the SCIUS in the SCEs of the application, and the evolution of the stability of the SCE that can be ascribed to the SCIUS. This phase is organized so that the same analysis of evolution is done for several effects of the SCIUS in the application. Therefore, each sub-phase only explains the effect that one wants to measure and the way to assess such effect.

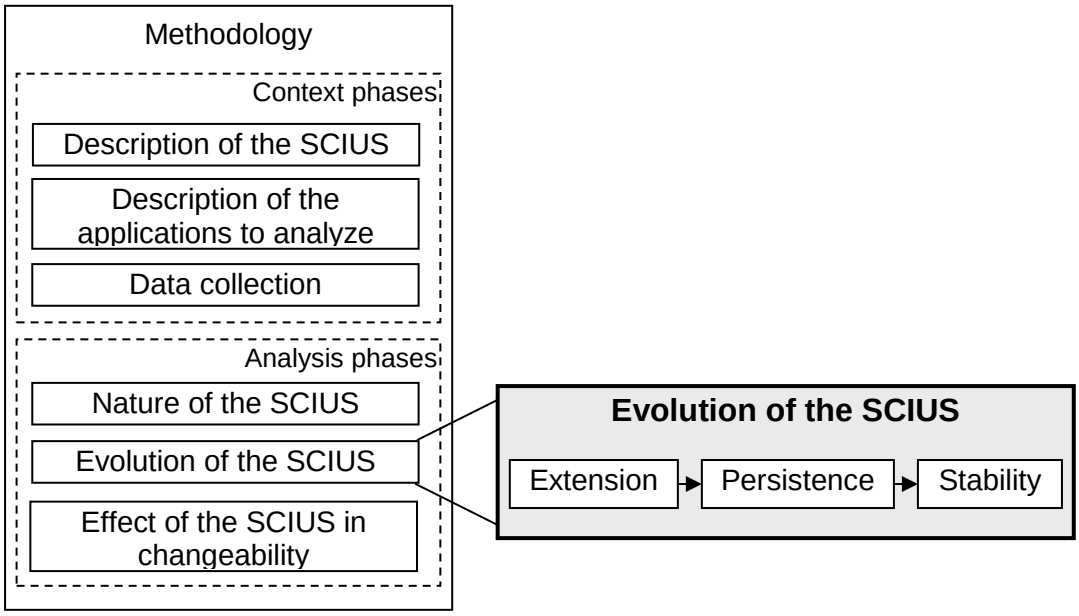


Figure 9-1. Description of the phase "Evolution of the SCIUS".

9.1 Phase description

This section of the chapter presents the steps needed to analyze the evolution of the SCIUS in the application, and in some cases, at the level of the SCE. Figure 9-2 presents the history of a fictitious application that is used as example to explain the formulas of the evolution of the

extension, persistence, and stability of the SCIUS. Appendix A contains a detailed explanation of the formulas presented.

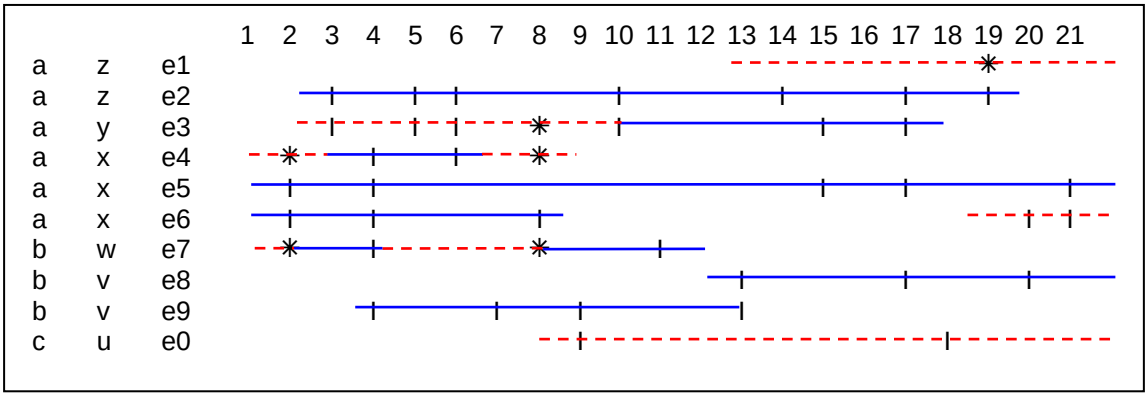


Figure 9-2. History of a fictitious application. Each row is a SCE, each column is a commit transaction. The vertical segments indicate a change in that SCE at that commit transaction. The asterisks indicate changes in the SCIUS. Red dashed lines indicate periods with the SCIUS. Blue straight lines indicate periods without the SCIUS

Deliverable: An analysis on the relation between the extension, persistence, and stability of the SCIUS and the changeability effects of the SCIUS.

Rationale: This set of analyses would state in a general way the magnitude of the supposed effects of the SCIUS on changeability. The magnitude is expressed in three ways, with the extension, the persistence, and the stability of the SCIUS. The extension measures the extent of the application that has been affected by the SCIUS. The persistence measures the average lifetime of SCIUS in the SCEs of the application. The stability measures the extent of changes in the application that can be directly linked to the SCIUS.

Procedure: Plot the value of the following functions for each commit transaction: ***extension_application***, ***persistence_application***, and ***stability_application***. These graphs would show if the effect of the SCIUS metric grows (line with a positive slope), shrinks (line with a negative slope) or remains stable over time (line without slope), as well as the rapidity of this behavior (linear, polynomial, logarithmic or exponential line). Repeat the analysis for the ***extension_SCE***, and ***stability_SCE*** to see the effect of the SCIUS at the level of SCEs.

Plot the value of ***extension_contribution***, ***persistence_contribution***, and

stability_contribution per modules or per the largest SCE analyzed. These graphs show how even is the measure of the SCIUS across the application, how large is the difference between the module most affected and the one least affected, and therefore to what extent the type of module can be linked to the SCIUS.

For each graph discuss if the pattern shown by the graph presents the worse, average, or best case scenario in terms of the effect of the SCIUS in the changeability of the application.

9.1.1 Extension

Extension is the percentage of SCEs with a SCIUS from the whole set of SCEs. It says to what extent an application is affected by the SCIUS. The graphs proposed to analyze extension are:

Extension per commit transaction: it is the average extension of the SCIUS in the application by the given commit transaction ***i***.

$$\mathbf{extension\_application} : \aleph \rightarrow \Re$$

$$\mathbf{extension\_application}(i) = \frac{\left\| \bigcup_{e \in Si} (\mathbf{hasSCIUS}(e, i)) \right\|}{\|S_i\|}$$

Equation 9-1. Extension application

The function ***extension_application*** gives the percentage of SCEs that have the SCIUS at a commit transaction ***i***. It is defined as the number of SCEs that had the SCIUS at the commit transaction ***i***, over the number of SCEs that compose the application at the commit transaction ***i*** (i.e. the size of the snapshot ***Si***). For instance the extension at the commit transaction 8 (see Figure 9-2) of the example application is

$$\mathbf{extension\_application}(8) = \frac{\|\{a.y.e3,\ a.x.e4,\ b.w.e7,\ c.u.e0\}\|}{\left\|\begin{Bmatrix} a.z.e2,\ a.y.e3,\ a.x.e4,\ a.x.e5, \\ a.x.e6,\ b.w.e7,\ b.v.e9,\ c.u.e0 \end{Bmatrix}\right\|} = \frac{4}{8} = 0.5$$

Therefore, 50% of the application has the SCIUS at the eighth commit transaction.

Contribution of a SCE of higher granularity than the one analyzed to the extension of SCIUS in the application: it is the extension of the SCIUS in the application, by the given commit transaction ***i***, due to the SCE analyzed (***e1***).

$$\mathbf{extension\_contribution} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{extension\_contribution}(i, e1) = \frac{\left\| \bigcup_{e2 \in e1} (\mathbf{hasSCIUS}\ (e2, i)) \right\|}{\left\| \bigcup_{e3 \in Si} (\mathbf{hasSCIUS}\ (e3, i)) \right\|}$$

Equation 9-2. Extension contribution

The function **extension_contribution** gives the percentage that a SCE **e1** contributes to the SCIUS in the application at the commit transaction **i**. It is defined as the division between the size of the set of SCEs that have the SCIUS at that commit transaction **i** and that compose the SCE of higher granularity **e1**. Over the size of the set of SCEs that have the SCIUS at that commit transaction **i**. In other words, the numerator is the amount of SCEs **e2** that have the SCIUS at the commit transaction **i**, and that belong to the SCE **e1**. The denominator is the amount of SCEs **e3** that have the SCIUS at the commit transaction **i**. For instance the contribution of the SCE **a.x** to extension of the SCIUS in the application at the commit transaction 8 (see Figure 9-2) is

$$\mathbf{extension\_contribution}\ (8, \mathbf{a.x}) = \frac{\| \{a.x.e4\} \|}{\|\{a.y.e3,\ a.x.e4,\ b.w.e7,\ c.u.e0\}\|} = \frac{1}{4} = 0.25$$

Therefore, the SCE **a.x** contributes to the 25% of the SCIUS of the application at the eighth commit transaction.

9.1.1.1 Extension in SCEs

Another possible definition for extension is the number of components of a SCE affected by the SCIUS, from all the components of the SCE. This definition is only valid in the case where the SCIUS is defined in terms of components of the SCE to be analyzed. For instance, if the SCE analyzed are classes, but the SCIUS occurs at the level of methods, e.g., long parameter list.

Extension by SCE: it is the average extension of the SCIUS inside the SCEs by the given commit transaction **i**.

$$\mathbf{extension\_SCE} : \aleph \rightarrow \Re$$

$$\mathbf{extension\_SCE}(\mathbf{i}) = \frac{\sum_{\mathbf{e} \in \mathbf{Si} \wedge (\mathbf{hasSCIUS}(\mathbf{e},\mathbf{i})=1)} \frac{\mathbf{size\_scius}(\mathbf{e},\mathbf{i})}{\mathbf{size\_SCE}(\mathbf{e},\mathbf{i})}}{\sum_{\mathbf{e} \in \mathbf{Si} \wedge (\mathbf{hasSCIUS}(\mathbf{e},\mathbf{i})=1)} 1}$$

Equation 9-3. Extension SCE

The function ***extension_SCE*** gives the average percentage of components of the SCE ***e*** that have the SCIUS at a commit transaction ***i***. The extension of the SCIUS in a SCE is defined as the number of components of the SCE ***e*** that have the SCIUS (***size_scius(e,i)***), over the number of components of the SCE ***e*** (***size_SCE(e,i)***). To calculate the average extension of the SCIUS in the SCEs of an application, it is enough to sum the extension of the SCIUS in all SCEs that have the SCIUS, and divide it over the number of SCEs that have the SCIUS. For instance, the average extension per SCE at the commit transaction 8 (see Figure 9-2) is

$$\mathbf{extension\_SCE}\ (8) = \frac{\frac{size\_scius\,(a.y.e3)}{size\_SCE(a\,.y.e3)} + \frac{size\_scius\,(a.x.e4)}{size\_SCE(a\,.x.e4)} + \frac{size\_scius\,(b.w.e7)}{size\_SCE(b\,.w.e7)} + \frac{size\_scius\,(c.u.e0)}{size\_SCE(c\,.u.e0)}}{4}$$

Supposing that all SCEs have the same size 50, and that the components affected by the SCIUS on *a.y.e3* is 30, on *a.x.e4* is 35, on *b.w.e7* is 40, and on*c.u.e0* is 45; the result would be

$$\mathbf{extension\_SCE}\ (8) = \frac{\frac{30}{50} + \frac{35}{50} + \frac{40}{50} + \frac{45}{50}}{4} = \frac{0.6 + 0.7 + 0.8 + 0.9}{4} = 0.75$$

Therefore, in average, 75% of the components of the SCEs are affected by the SCIUS.

9.1.2 Persistence

Persistence is the percentage of the lifetime of a SCE that includes a SCIUS. It aims to show the longevity of the SCIUS in the application. The graphs proposed to analyze it are:

Persistence per commit transaction: it is the average lifetime percentage of the SCEs with the SCIUS by the given commit transaction ***i***.

$$persistence\_application : \aleph \rightarrow \Re$$

$$persistence\_application(i) = \frac{\sum_{e \in SCE\_affected\_by\_SCIUS(i)} persistence\_SCE(e, i)}{\|SCE\_affected\_by\_SCIUS(i)\|}$$

Equation 9-4. Persistence application

The function ***persistence_application*** gives the average persistence of SCEs affected by the SCIUS at the given commit transaction ***i***. It is defined as the division between the result of the function ***persistence_SCE*** over the size of the set that results from the function ***SCE_affected_by_SCIUS***.

The function ***SCE_affected_by_SCIUS*** gives a set of SCEs that have been affected by the SCIUS by the given commit transaction ***i***.

The function ***persistence_SCE*** gives the percentage of commit transactions of the lifetime of the SCE ***e***, that it has had the SCIUS by the given commit transaction ***i***. It is defined as the division between the result of the function ***commits_with_SCIUS*** over the result of the function ***commits_alive*** . The function ***commits_with_SCIUS*** counts the number of commit transactions that a SCE ***e*** have had the SCIUS by the given commit transaction ***i***. The function ***commits_alive*** counts the number of commit transactions that the SCE ***e*** has been part of the system by the given commit transaction ***i***.

For instance, the average persistence of the SCIUS of the example (see Figure 9-2) at the commit transaction 8 is:

$$persistence\_application(8) = \frac{\begin{array}{l} persistence\_SCE(a.y.e3, 8) \\ + persistence\_SCE(a.x.e4, 8) \\ + persistence\_SCE(b.w.e7, 8) \\ + persistence\_SCE(c.u.e0, 8) \end{array}}{\|\{a.y.e3,\ a.x.e4,\ b.w.e7,\ c.u.e0\}\|}$$

Which is equivalent to:

$$\mathbf{persistence\_application}(8) = \frac{\begin{array}{l}\mathbf{commits\_with\_SCIUS}(a.y.e3,8)/\mathbf{commits\_alive}(a.y.e3,8) \\ + \mathbf{commits\_with\_SCIUS}(a.x.e4,8)/\mathbf{commits\_alive}(a.x.e4,8) \\ + \mathbf{commits\_with\_SCIUS}(b.w.e7,8)/\mathbf{commits\_alive}(b.w.e7,8) \\ + \mathbf{commits\_with\_SCIUS}(c.u.e0,8)/\mathbf{commits\_alive}(c.u.e0,8)\end{array}}{\| \{a.y.e3,\ a.x.e4,\ b.w.e7,\ c.u.e0\} \|}$$

That is,

$$\mathbf{persistence\_application}(8) = \frac{7/7 + 4/8 + 5/8 + 1/1}{4} = \frac{3.12}{4} = 0.78$$

Therefore, in average by the eighth commit, the SCIUS of the application tended to last 78% of the lifetime of the SCEs.

Persistence of a SCE of higher granularity level than the one analyzed: it is the average persistence of the SCEs that form the given SCE analyzed (***e1***) by the given commit transaction ***i***.

$$\mathbf{persistence\_contribution} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{persistence\_contribution}(\mathbf{i}, \mathbf{e}1) = \frac{\sum_{\mathbf{e}2 \in \mathbf{e}1 \wedge \mathbf{e}2 \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} \mathbf{persistence\_SCE}(\mathbf{e}2, \mathbf{i})}{\sum_{\mathbf{e}2 \in \mathbf{e}1 \wedge \mathbf{e}2 \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} 1}$$

Equation 9-5. Persistence contribution

The function ***persistence_contribution*** gives the average persistence of the SCEs ***e2*** that compose the SCE ***e1*** by the commit transaction ***i***. The function ***SCE_affected_by_SCIUS*** gives a set of SCEs that have been affected by the SCIUS by the given commit transaction ***i***. That is, the average persistence of the SCEs ***e2*** that compose the SCE ***e1*** (i.e. the average persistence of the ***e2*** ∈ ***e1***). For instance the contribution of the SCE *a.x* to persistence of the SCIUS in the application at the commit transaction 8 (see Figure 9-2) is

$$persistence\_contribution(8, a.x) = \frac{persistence\_SCE(a.x.e4, 8)}{\|\{a.x.e4\}\|}$$

$$persistence\_contribution(8, a.x) = \frac{\frac{commits\_with\_SCIUS(a.x.e4, 8)}{commits\_alive(a.x.e4, 8)}}{\|\{a.x.e4\}\|}$$

$$persistence\_contribution(8, a.x) = \frac{\frac{4}{8}}{1} = 0.5$$

Therefore, the average persistence of the SCEs that belong to $a.x$ at the eighth commit transaction is 50% of their lifetime.

9.1.3 Stability

Stability is the percentage of changes to a SCE that occur inside the SCIUS, it aims to show the variability that the SCIUS introduces in the application. The graphs proposed to analyze stability are:

Stability by history point: it is the average stability of the SCEs in the application while having the SCIUS by the commit transaction i.

$$stability\_application : \aleph \rightarrow \Re$$

$$stability\_application(i) = \frac{\sum_{e \in SCE\_affected\_by\_SCIUS(i)} stability\_SCE(e, i)}{\| SCE\_affected\_by\_SCIUS(i) \|}$$

Equation 9-6. Stability application

The function $stability\_application$ gives the average stability of SCEs affected by the SCIUS at the given commit transaction i. It is defined as the division between the result of the function $stability\_SCE$ over the size of the set that results from the function $SCE\_affected\_by\_SCIUS$. The function $SCE\_affected\_by\_SCIUS$ gives a set of SCEs that have been affected by the SCIUS by the given commit transaction i. The function $stability\_SCE$ gives a set of SCEs that have been affected by the SCIUS by the given commit transaction i. The function $changes\_by$ gives the set of commit transactions in which the SCE e is modified until the given commit transaction i. The function $changes\_when\_SCIUS$ gives the set of commit transactions in which the SCE e is modified,

having the SCIUS, by the given commit transaction ***i***.

For instance, the average stability of the SCIUS of the example (see Figure 9-2) at the commit transaction 8 is:

$$\mathbf{stability\_application}(8) = \frac{\begin{array}{l}\mathbf{stability\_SCE}(a.y.e3,8)\\ +\ \mathbf{stability\_SCE}(a.x.e4,8)\\ +\ \mathbf{stability\_SCE}(b.w.e7,8)\\ +\ \mathbf{stability\_SCE}(c.u.e0,8)\end{array}}{\| \{a.y.e3,\ a.x.e4,\ b.w.e7,\ c.u.e0\} \|}$$

Which is equivalent to:

$$\mathbf{stability\_application}(8) = \frac{\begin{array}{l}\mathbf{changes\_when\_SCIUS}(a.y.e3,8) \,/\, \mathbf{changes\_by}(a.y.e3,8)\\ +\ \mathbf{changes\_when\_SCIUS}(a.x.e4,8) \,/\, \mathbf{changes\_by}(a.x.e4,8)\\ +\ \mathbf{changes\_when\_SCIUS}(b.w.e7,8) \,/\, \mathbf{changes\_by}(b.w.e7,8)\\ +\ \mathbf{changes\_when\_SCIUS}(c.u.e0,8) \,/\, \mathbf{changes\_by}(c.u.e0,8)\end{array}}{\| \{a.y.e3,\ a.x.e4,\ b.w.e7,\ c.u.e0\} \|}$$

That is,

$$\mathbf{stability\_application}(8) = \frac{4/4 + 2/4 + 2/3 + 0/0}{4} = \frac{2.16}{4} = 0.54$$

Therefore, in average by the eighth commit, 54% of the changes on the SCEs tended to occur whenever the SCEs had the SCIUS.

Contribution of SCE of higher granularity than the one analyzed to the stability of the application: it is the average stability of the SCEs that compose the given SCE ***e1*** by the given commit transaction ***i***.

$$\mathbf{stability\_contribution} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{stability\_contribution}(\mathbf{i}, \mathbf{e}1) = \frac{\sum_{\mathbf{e}2 \in \mathbf{e}1 \wedge \mathbf{e}2 \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} \mathbf{stability\_SCE}(\mathbf{e}2, \mathbf{i})}{\sum_{\mathbf{e}2 \in \mathbf{e}1 \wedge \mathbf{e}2 \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} 1}$$

Equation 9-7. Stability contribution

The function ***stability_contribution*** gives the percentage that a SCE ***e1*** contributes to the stability of the SCIUS in the application by the commit transaction ***i***. That is, the average stability of the SCEs ***e2*** that compose the SCE ***e1*** (i.e. the average stability of the ***e2*** $\in$ ***e1***). For instance the contribution of the SCE ***a.x*** to persistence of the SCIUS in the application at the commit transaction 8 (see Figure 9-2) is

$$\mathbf{stability\_contribution}\ (8, \mathbf{a.x}) = \frac{\mathbf{stability\_SCE}(a.x.e4, 8)}{\|\{a.x.e4\}\|}$$

$$\mathbf{stability\_contribution}\ (8, \mathbf{a.x}) = \frac{\frac{\mathbf{changes\_when\_SCIUS}(a.x.e4, 8)}{\mathbf{changes\_by}(a.x.e4, 8)}}{\|\{a.x.e4\}\|}$$

$$\mathbf{stability\_contribution}\ (8, \mathbf{a.x}) = \frac{2/4}{1} = 0.5$$

Therefore, by the 8th commit transaction, 50% of the changes in SCEs that belong to ***a.x*** occurred while having the SCIUS.

9.1.3.1 Stability in SCEs

Another possible definition for stability is the number of changes in the SCE inside the components of the SCE affected by the SCIUS, from all the changes of the SCE when it has the SCIUS. This definition aims to assess the percentage of changes due to the SCIUS. This definition is only valid in case the SCIUS is defined in terms of components of the SCE to be analyzed.

Stability by SCE: it is the average stability of the SCE due to the SCIUS by the given commit transaction ***i***.

The function ***stability_SCE*** gives the average percentage of changes that have occurred in the components of ***e*** that have the SCIUS by the commit transaction ***i***. That is, the number of changes of the SCE ***e*** that have occurred inside the SCIUS (***changes_in_SCIUS***), over the number of changes on the SCE ***e*** while it has the SCIUS (***changes_when_SCIUS***). The function ***changes_in_SCIUS*** gives the set of commit transactions in which any components

of the SCE **e** with the SCIUS have been modified, by the given commit transaction **i**. The function **changes_when_SCIUS** gives the set of commit transactions in which the SCE **e** is modified, having the SCIUS, by the given commit transaction **i**.

$$\mathbf{stability\_SCE} : \aleph \rightarrow \Re$$

$$\mathbf{stability\_SCE}(\mathbf{i}) = \frac{\sum_{\mathbf{e} \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} \frac{\left\|\mathbf{changes\_in\_SCIUS}(\mathbf{e},\mathbf{i})\right\|}{\left\|\mathbf{changes\_when\_SCIUS}(\mathbf{e},\mathbf{i})\right\|}}{\sum_{\mathbf{e} \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} 1}$$

Equation 9-8. Stability SCE

For instance, the average stability per SCE at the commit transaction 8 (see Figure 9-2) is

$$\mathbf{stability\_SCE}(8) = \frac{\begin{array}{l}\frac{\left\|\mathbf{changes\_in\_SCIUS}(a.y.e3)\right\|}{\left\|\mathbf{changes\_when\_SCIUS}(a.y.e3)\right\|} \\ + \frac{\left\|\mathbf{changes\_in\_SCIUS}(a.x.e4)\right\|}{\left\|\mathbf{changes\_when\_SCIUS}(a.x.e4)\right\|} \\ + \frac{\left\|\mathbf{changes\_in\_SCIUS}(b.w.e7)\right\|}{\left\|\mathbf{changes\_when\_SCIUS}(b.w.e7)\right\|} \\ + \frac{\left\|\mathbf{changes\_in\_SCIUS}(c.u.e0)\right\|}{\left\|\mathbf{changes\_when\_SCIUS}(c.u.e0)\right\|}\end{array}}{4}$$

That is,

$$\mathbf{stability\_SCE}(8) = \frac{\frac{\|\{8\}\|}{\|\{3,5,6,8\}\|} + \frac{\|\{2,8\}\|}{\|\{2,8\}\|} + \frac{\|\{2,8\}\|}{\|\{2,8\}\|} + \frac{\|\{\}\|}{\|\{\}\|}}{4} = \frac{\frac{1}{4} + \frac{2}{2} + \frac{2}{2} + 0}{4} = 0.56$$

Therefore, by the 8th commit transaction, in average, 56% of the changes in SCEs occurred inside the SCIUS.

9.2 Phase application

This section presents the results of tracking the extension, persistence, and stability of cloning in the application over the application's lifetime. A detailed explanation on the calculation of the extension, persistence and stability is shown in section 9.1.

9.2.1 Extension of clones in the application

Here we present the percentage of cloning found in the applications analyzed. The results on the percentage of cloning presented in the literature have been diverse: there are authors that report just 6% of the application [Lague '97], while others have found up to 50% [Ducasse '99] of the application cloned. However, most of the results point out to a level of 10% to 20% of the application [Baker '95; Mayrand '96; Baxter '98; Kapser '06b]. Based on those results we expected that cloning in the applications analyzed would not be above 20%, and that would remain stable over time.

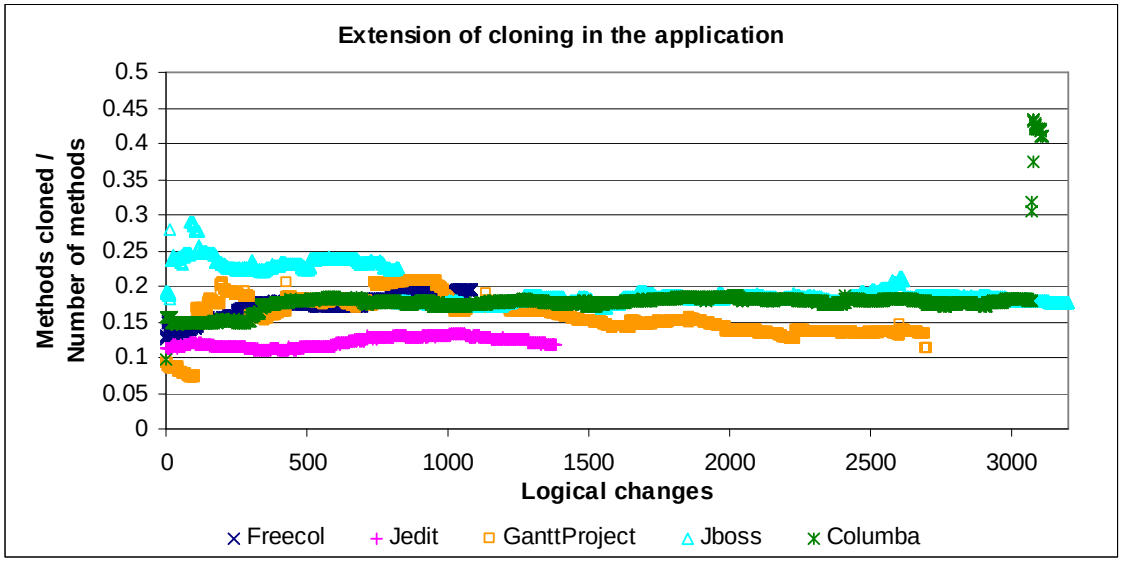


Figure 9-3. Extension of cloning in the applications analyzed

Figure 9-3 presents the results of the formula ***extension_application*** presented in section 9.1.1 for all the commit transactions of the applications analyzed. The x-axis has the commit transactions in sequential order, and the y-axis has the results of the extension of cloning in the methods of the application for that commit transaction. We have found that the percentage of cloning in the application is between 10% and 20% of the application; and that it tends to remain stable over time, see Figure 9-3.

For the most part the curve representing the extension of cloning is continuous; however, there

are some discontinuities in the curves of Ganttproject, JBoss, and Columba. The discontinuities in Ganttproject are related with deletion of a large number of methods. If most of the methods deleted are cloned, the curve falls. If most of the methods deleted are not cloned, the curve rises. The discontinuities in JBoss are related with the addition of a large amount of methods at once, from which a high percentage is cloned. In particular, the curve jumps up (commit 87) when adding a package to handle pools of objects from which 38% of its methods are cloned. Conversely, the curve falls when that package is replaced by a jar that will be maintained by a third party (commit 825). The largest discontinuity is found in Columba, by the end of the analyzed history. In these last commits, the developers are restructuring the application to migrate from CVS to SVN. In the commit 3072, the number of methods starts to grow, as well as the number of cloned methods. Judging by the messages of the last commits, they copied most of the code in a different directory structure, while at the same time, they were fixing warnings, updating the documentation, standardizing the format of the code, and adding some key functionality.

Notice in Figure 9-3 that the curves corresponding to the applications are very similar to each other regardless of the differences in size, number of developers, and age of the applications. Given that the size of the applications increases over time, cloning increases in a proportional rate. It does not seem to be a policy of clone removal in any of the applications analyzed, because, most of the changes in the extension of cloning are due to accidental modification of the ratio between cloned and not cloned methods.

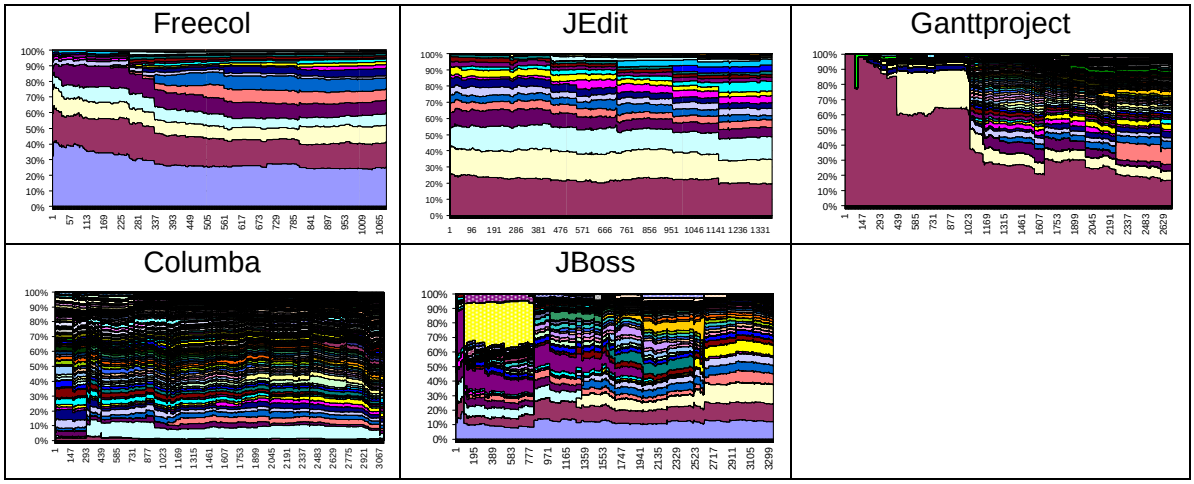


Figure 9-4. Contribution of each package to the cloning of the applications.

Figure 9-4 presents the percentage of cloning that each package gives to the overall cloning for

each application. That is, the results of the formula ***`extension_contribution`*** (presented in section 9.1.1) for all the commit transactions of each one of the applications analyzed. The x-axis has the commit transactions in sequential order, and the y-axis has the percentage of cloning of each package at that commit transaction. The percentage of cloning is calculated as the number of methods with clones in that package over the number of methods in the whole application. Each package is represented by a different color; the width of a package's color stripe indicates the percentage of cloning that the package contributes to the cloning of the complete application. At first glance, it is evident that the way in which cloning is distributed across the application varies depending on the application. Freecol and JEdit have a few packages that are the main contributors to the cloning in the application, while Columba's cloning seems to be evenly distributed across packages. Although Columba has a main package that contributes to a large proportion of the clones in the application, note that the contribution of this package is less minor (between 0.6% and 9%) compared to the package that contributes the most to any of the other applications. The package that contributes the most in any of the other applications accounts for at least 19% of their clones. This difference might be due to the different level of encapsulation of the applications: while Freecol has 28 packages, JEdit has 31, Ganttproject has 116, Columba has 346, and JBoss has 141. This hypothesis is supported by the behavior of Ganttproject. At the beginning of the history of Ganttproject, all the cloned methods are in the package `net.sourceforge.ganttproject`, this is because all the methods are located in that package. Over time, the application is gradually restructured, which also shifts the distribution of the clones.

Note that, in Figure 9-4, a package that is an important contributor of the cloning in JBoss disappears by commit 777 (shown in yellow at the top left hand side of Figure 9-4). This package is 'minerva.jdbc' that in that commit was replaced by a jar that is managed by a third party.

Table 9-1. Packages that contributed the most to cloning on each application.

Application	Packages that contributed the most to cloning in descending order
Freecol	net.sf.freecol.common.model net.sf.freecol.client.gui.panel net.sf.freecol.client.control net.sf.freecol.server.control net.sf.freecol.client.gui
JEdit	org.gjt.sp.jedit org.gjt.sp.jedit.textarea org.gjt.sp.jedit.gui org.gjt.sp.jedit.options org.gjt.sp.jedit.search
Ganttproject	net.sourceforge.ganttproject net.sourceforge.ganttproject.gui org.jdesktop.swingx net.sourceforge.ganttproject.io org.ganttproject.ics.parser
Columba	org.columba.mail.pop3 org.columba.core.scripting org.columba.mail.gui.table.action org.columba.mail.imap org.columba.mail.gui.composer.html.action
JBoss	org.jboss.ejb org.jboss.ejb.plugins org.jboss.ejb.plugins.cmp.jdbc org.jboss.ejb.deployment org.jboss.ejb.plugins.jaws.deployment

Code that implements concerns related to I/O, and GUI in Java are believed to be more susceptible of being cloned due to the nature of the API provided by Java [Kim '04]. Therefore, we expected packages that contributed more to the cloning of the application to be related with these concerns. Although these packages were among those that contributed the most, we found surprising that in all applications packages belonging to the core of the application are also highly cloned. Table 9-1 summarizes the top five packages that contributed the most to cloning in each one of the applications analyzed (in order of contribution).

9.2.1.1 Extension in methods

Some authors have tried to measure the extension of cloning at a finer level of granularity than methods. For instance, by measuring the percentage of statements cloned in a subsystem [Baxter '98; Ducasse '06], or the percentage of LOCs cloned in a module [Monden '02]. These measurements do not show at which level abstractions are cloned. In the case of lines of code

cloned in a module, it is not clear whether the cloned lines represent a structure, a method, or a class. Therefore, we decided to measure the extension of cloning inside methods and to track it over time. Results are shown in Figure 9-5. This figure presents in the x-axis the commit transactions in sequential order, and the y-axis has the results of applying the formula ***extension_SCE*** for each commit transaction on the x-axis. The formula indicates the average percentage of tokens that are part of a cloned fragment of the tokens in methods with clones (see section 9.1.1).

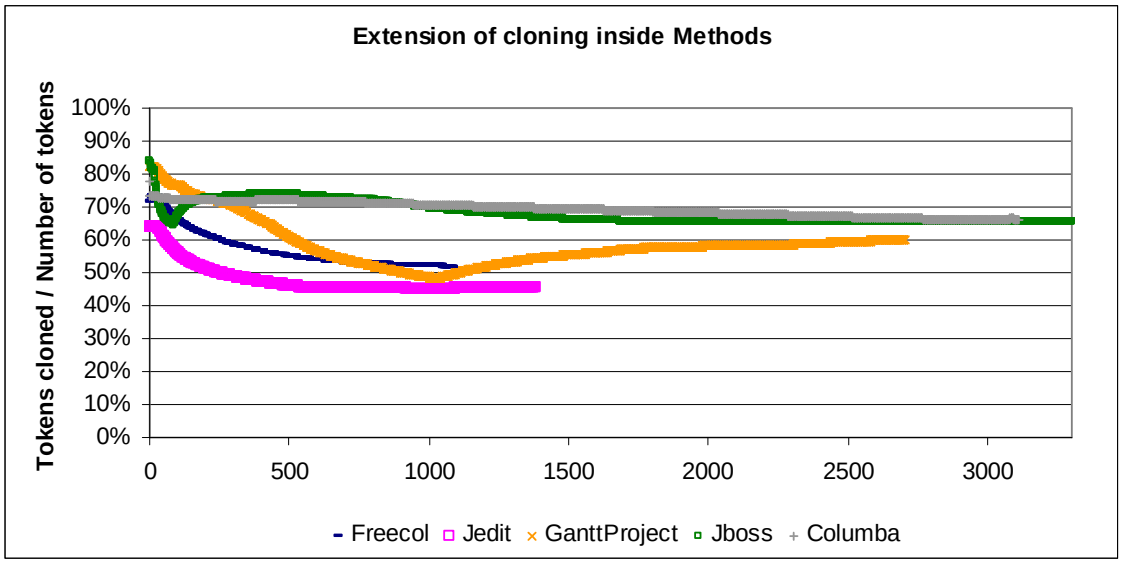


Figure 9-5. Extension of cloning inside methods.

We have found that whenever methods are cloned, they tend to be highly cloned; in particular, for large applications. Over time, the percentage of cloning decreases, probably due to segmentation of the cloned fragments. Note that, until the commit 1050 there is a rapid decrease in the percentage of cloning per method in Ganttproject. From this commit onward, several test classes are added. The addition of test methods that are mostly cloned is the reason for the slow increase of the average of cloning inside methods. Similarly, with the inflection point in JBoss at commit 87, several cloned methods were added, increasing again the average.

The slow decrease of the percentage cloned in the large applications occurs because their cloned fragments are less changed than the cloned fragments of smaller applications. We have found that the average number of changes in the clone per pair of methods was 9.5, 8.5, and 8.4, for Freecol, JEdit and Ganttproject respectively. However, the average number of changes in the clone per pair of methods was just 4.1 for Columba and 5 for JBoss. The lower amount of

changes in the clones of Columba and JBoss is not due to less change in those applications given that all applications have the same amount of average changes per method, which are two changes per method. The lower amount of changes in the clones of Columba and JBoss might be explained by their large amount of exact clones. The percentage of exact families of the families in the applications analyzed is 19% for Freecol, 25% for JEdit, 37% for Gantt project, 54% for Columba, and 28% for JBoss. This hypothesis is supported by the fact that before the inflection points of the extension curves, Ganttproject only had 42%, and JBoss 19% of the exact families they present across their history. Confirming the relationship between exact clones and the decay in cloning extension per method is left for future work.

9.2.2 Persistence of clones in the application

In [Kim '05] it is suggested that persistent clones are those that cover the lack of abstraction mechanisms in the language (such as aspects), those that are inevitable (such as API typical usage or paired operations), and that the rest of the cloned fragments tend to have a short lifetime. However, regardless of the nature of the clone families, our results in the nature of clones suggest that whenever a method has a cloned fragment, it will remain cloned most of its lifetime. In this section, the average persistence of clones in methods is evaluated after each change of the application, where the persistence of clones in a method is the percentage of commits that the method has had a clone. In this way, it is possible to see the progression of cloning persistence over time.

Figure 9-6 has in the x-axis has the commit transactions in sequential order, and the y-axis has the results of the formula ***persistence_application*** for each commit transaction. This means that Figure 9-6 shows the average persistence of cloning calculated after each change of the applications (see section 9.1.2).

The results are consistent with the results about the nature of cloning (see section 8.2.8), i.e. in average, methods are cloned at least 85% of their lifetime. Given the zoom in the y-axis they do not seem very similar. However, note that they are all in the same range, are rather stable with the exception of some events on each curve. All the curves tend to stabilize between .9 and .95 regardless of events of sudden increase or sudden reduction. Nevertheless, there are some large discontinuities in Freecol, Ganttproject, and Columba. The discontinuity in Freecol seems to be related with large amounts of clones inserted into preexistent methods, in the commit 803, 27

old methods were cloned. The one in Ganttproject in commit 427 happened because 43 methods with clones are created. As for Columba, the jump at the end occurs because the application is being restructured by copying and pasting methods from the current code-base into other directories. In general, the jumps up occur whenever several methods are created with clones; while the falls occur whenever several clones are introduced to pre-existing methods.

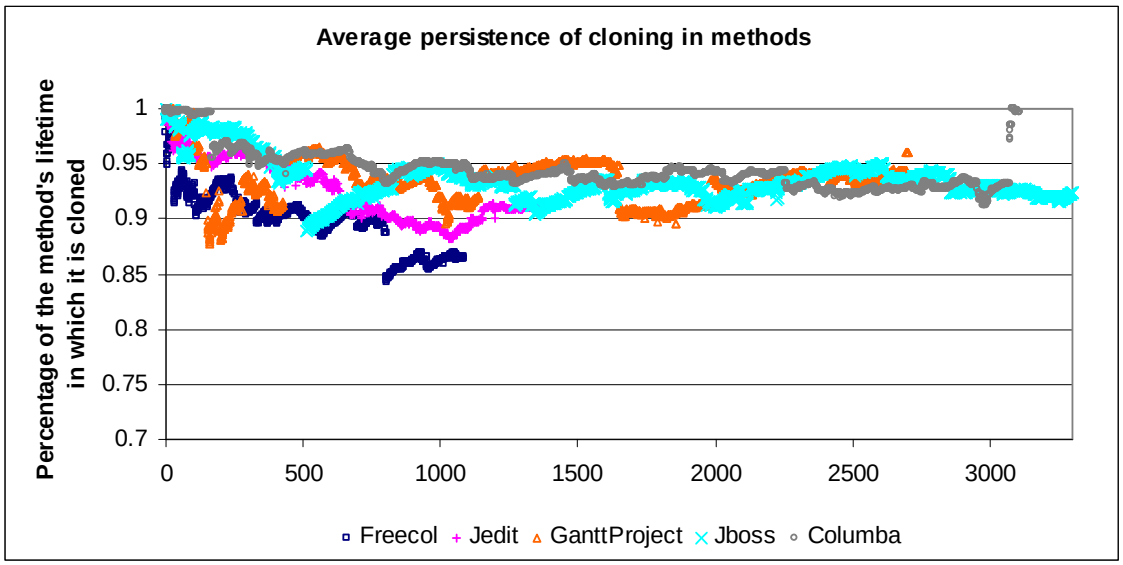


Figure 9-6. Average persistence of cloning in methods

We also wanted to verify if the persistence of the clones behave similarly across packages or if the clones on some packages were more persistent than the clones on other packages. Figure 9-7,Figure 9-8,Figure 9-9,Figure 9-10, and Figure 9-11 present the results for Freecol, JEdit, Ganttproject, Columba, and JBoss respectively. Each graph presents on the x-axis the commit transactions, and on y-axis the persistence of cloning on each package at that commit transaction. That is, the average persistence of cloning in the methods of the package after each change of the application (i.e. the results of the formula ***persistence_contribution*** defined in section 9.1.2). Given that most of the packages have a cloning persistence of 100%, we decided only to show those with low persistence, i.e. the packages whose clones last less than the methods where they are located. For clarity, only the 15 packages with lowest persistence of each application have a curve in the graph. The x-axis marks the commit transaction or logical change, and at the y-axis marks the persistence of cloning of a package at that moment in time. The rest of this section presents the persistence per packages for all the

applications analyzed, describing the inflection points and summarizing the patterns found from analyzing these events.

Figure 9-7 presents the packages with lowest persistence of Freecol. Notice all the packages identified to have a large percentage of the clones of the application (see Table 9-1) are also this graph. Note that the value for these packages is high, and that it usually decreases slowly. Other packages like `net.sf.freecol` present a stable beginning, followed by a rapid decrease, and ended by a slow increase. These packages have several clones deleted at the commit in which their stable period finishes, but also present a large addition of new methods with clones at the commit in which their decrease period finishes.

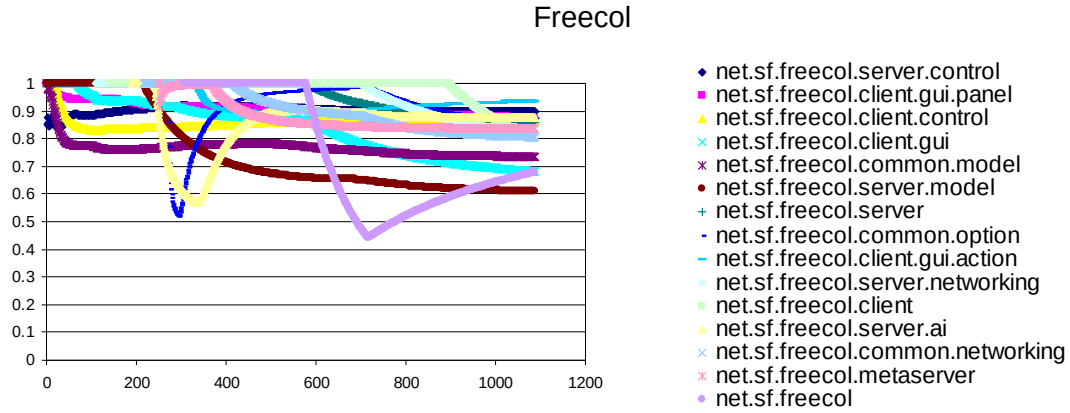


Figure 9-7. Evolution of persistence in Freecol packages with the lowest persistence.

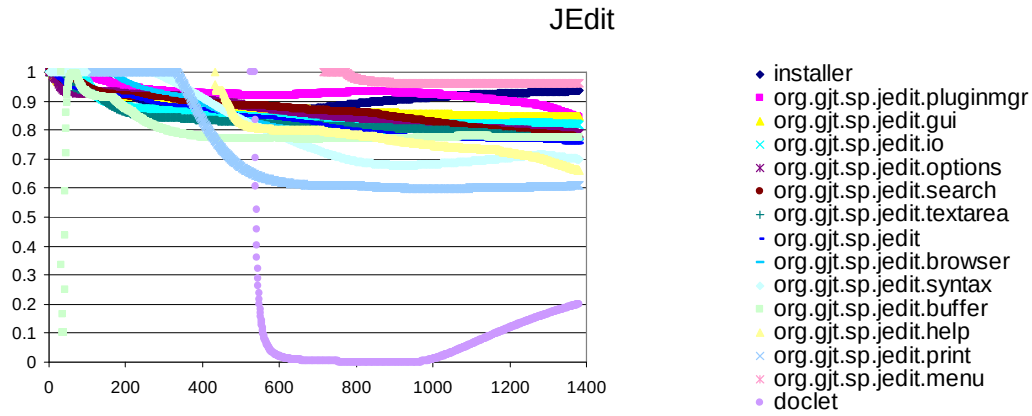


Figure 9-8. Evolution of persistence in JEdit packages with the lowest persistence.

Figure 9-8 presents the lowest persistence packages for JEdit. Note that in this graph are also present the packages with higher percentage cloned (see Table 9-1). As in Freecol, the

predominant sequence of patterns is stable persistence followed by a slow decay. Only two packages do not follow the main persistence decay pattern: they are `doclet` and `org.gjt.sp.jedit.buffer`. Although `org.gjt.sp.jedit.buffer` presents the predominant sequence of patterns from the commit 75, its beginning is a fast persistence increase. The package starts to be cloned in the commit 32 with four cloned methods, which are deleted in its second commit cloned. Therefore, at the beginning of the period cloned of the package, the only methods that have had clones were cloned only for one commit. However, the overall cloning-persistence of the package increases as more methods with clones are introduced to the package. The package `doclet`, in contrast, presents a period of decrease, followed by a period without clones, and ended by a slow increase. This package has 3 methods, 2 of them cloned, but one of the cloned methods is deleted after 10 commits; so the persistence drops rapidly until commit 995 in which another method of the package becomes cloned. By the end of the period analyzed that method has been cloned just 20% of its lifetime.

Figure 9-9 shows the packages with lowest cloning persistence in Ganttproject. In this graph appear only two of the five packages with higher percentage of clones (see Table 9-1). Therefore, it is not possible to establish any relation between the percentage cloned and the persistence of those clones. Note also that the persistence of most of the packages is stable at the beginning of their lives, followed by a slow decrease caused by eliminating the clones without eliminating the methods that had them. There are only two packages presenting an addition of clones, after the decrease caused by the deletion of clones from some methods.

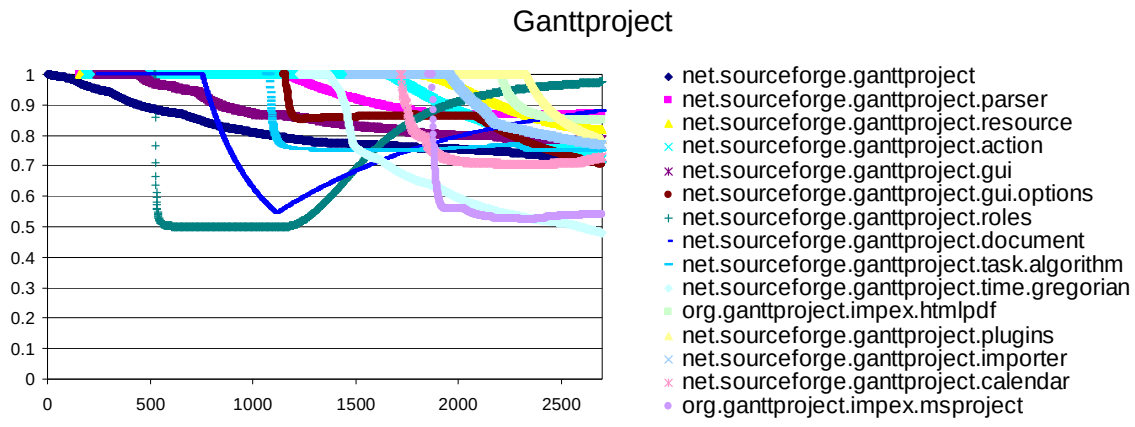


Figure 9-9. Evolution of persistence in Ganttproject packages with the lowest persistence.

Figure 9-10 shows the packages with lowest cloning persistence in Columba. None of the packages highly cloned is in this graph (see Table 9-1). This indicates that highly cloned

packages in Columba are more persistent. Note also that at least eight of the packages shown presents a clear addition of clones after the initial decrease of persistence. When looking at the messages of the commits that were the point of inflection in the curves, we found that in these commits, those packages were modified to use code that is inherently cloned such as the use of `iterators`, or the use of `internationalization` (i18n).

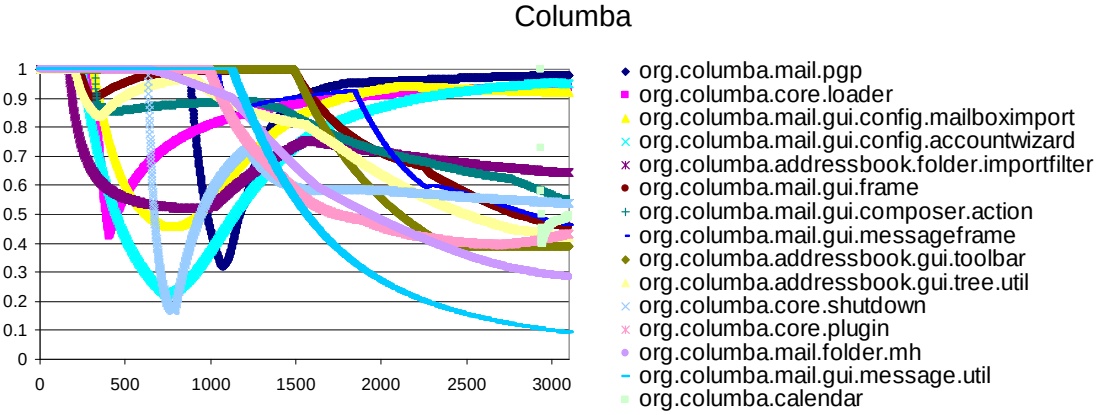


Figure 9-10. Evolution of persistence in Columba packages with the lowest persistence.

Figure 9-11 presents the packages with lowest persistence in JBoss. None of the packages with highest percentage of cloning had also low persistence for this application. This would mean that most of the cloned in JBoss persist all the lifetime of the methods that host them. Finally, note that, whenever there are increases, the increases of persistence are rather slow (low slope) and small (positive slope for a short period). A slow increase would mean that the addition of new clones is scarce, or that clones are mostly added to old methods. A small increase would mean that the percentage of clones added is small compared with the percentage of existing clones in the package.

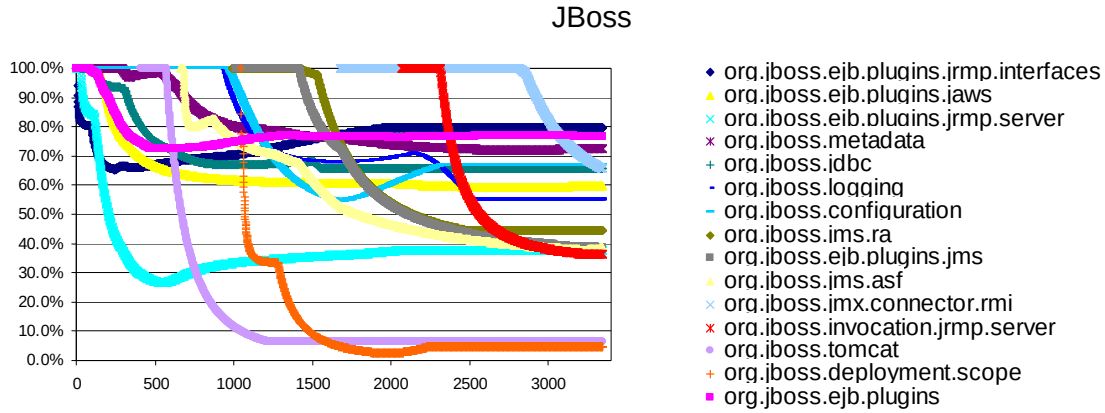


Figure 9-11. Evolution of persistence in JBoss packages with lowest persistence.

From the applications analyzed, we identified four patterns of evolution in the persistence of packages (see Figure 9-12).

The first pattern is whenever the persistence is stable at 100%. This pattern means that no clones have been deleted and that the methods with clones are introduced. This is the most common pattern and most of the packages presented it along all their lifetime.

The second pattern, which is the second most common, is characterized by slow decrease of the persistence. This pattern occurs when some of the clones of the package were deleted, without deleting the methods that hosted the clones. This is the most common pattern among the packages with lowest persistence.

The third pattern occurs when there is a slow increase of the persistence curve. Such slow increase is characterized by having a positive slope, usually of less of 45 degrees. This pattern is seen when pre-existing methods become cloned.

The fourth pattern occurs when there is a fast increase in the persistence, i.e. a curve with a very high slope. This pattern means that several methods with clones were introduced to the package.

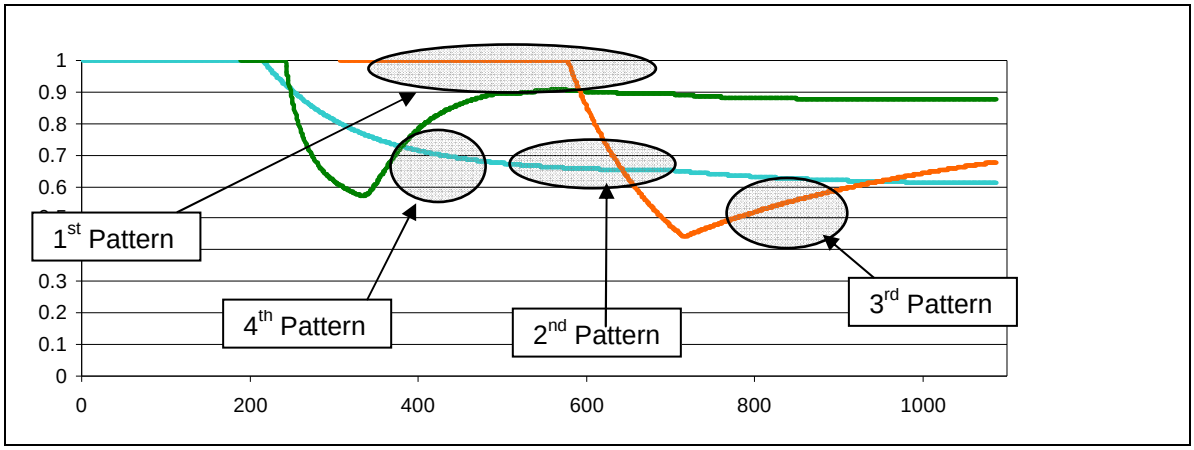


Figure 9-12. Patterns of evolution of persistence in packages.

In conclusion, most of the cloned methods remain cloned most of their lifetime; in some occasions, the packages with highest percentage cloned are also the ones with more persistent clones; and clones cover most of the lifetime of the method because they are rarely deleted without deleting the method that hosts them.

9.2.3 Stability of clones in the application

Finally, we check if clones are related with instability in the methods where they are placed, so we measure the accumulated number of changes in cloned methods over the accumulated number of changes in the whole application after each change of each application. The results of this analysis for all the applications analyzed are shown in Figure 9-13, which presents the results of the formula ***stability_application*** (section 9.1.3). The x-axis on Figure 9-13 has the commit transactions in sequential order, and the y-axis has the results of the instability in the methods of the application due to cloning until that commit transaction.

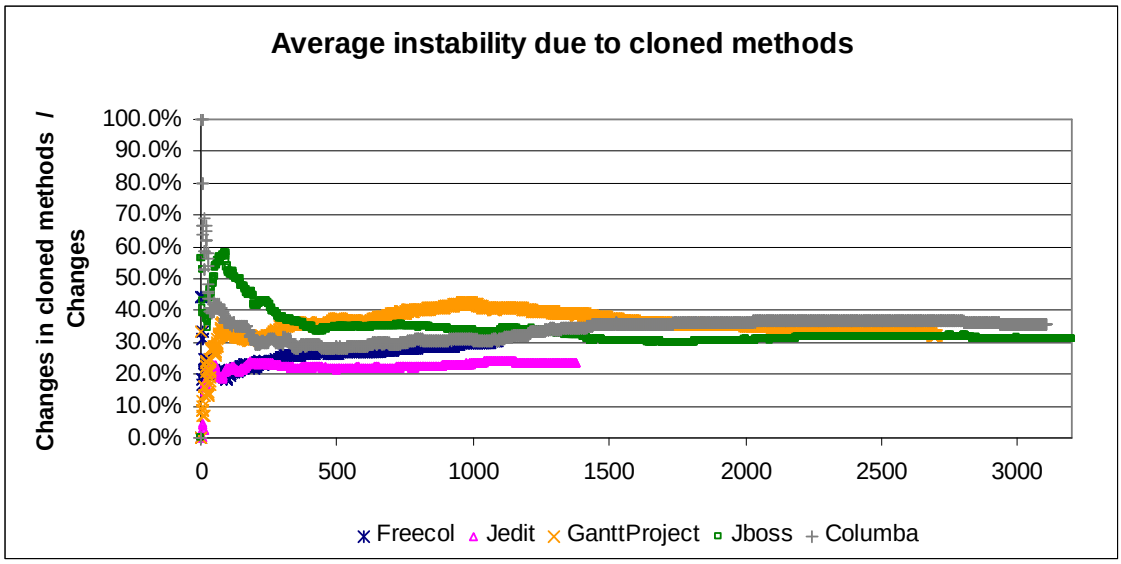


Figure 9-13. Average instability in the applications analyzed due to cloned methods

In section 9.1.1, we showed that the percentage of cloned methods in the applications analyzed lay between the 10% and 20% of the application. Figure 9-13 shows that from all the changes that occur in the application more than 20% occur when methods are cloned. Hence, we conclude that cloned methods tend to have a larger number of changes than methods without clones, 20% of the methods (i.e. those cloned) cause 40% of the changes. Note that, although the instability of methods with clones tends to vary significantly at the beginning of the application's history, the instability ratio tends to stabilize afterwards. Although most of the applications have just 20% of their methods cloned most of their lifetimes, the changes in these methods represent more than 20 to 30% of the changes in the case of Freecol and JEdit, and between 30% and 40% in the case of Ganttproject, Columba, and JBoss. In the latter applications, cloned methods have a large percentage cloned, and are cloned for long periods. As a result, it is likely that if these cloned methods change, their changes occur inside the cloned fragments. Note that this does not prove a causality relation between being cloned and changing. For instance, a larger number of changes when cloned might be due to cloned periods larger than not cloned periods. Another example is a larger extension of clones inside methods, because it increases the chance of modifying the cloned fragment, even though the change is unrelated to the semantics of the clone family.

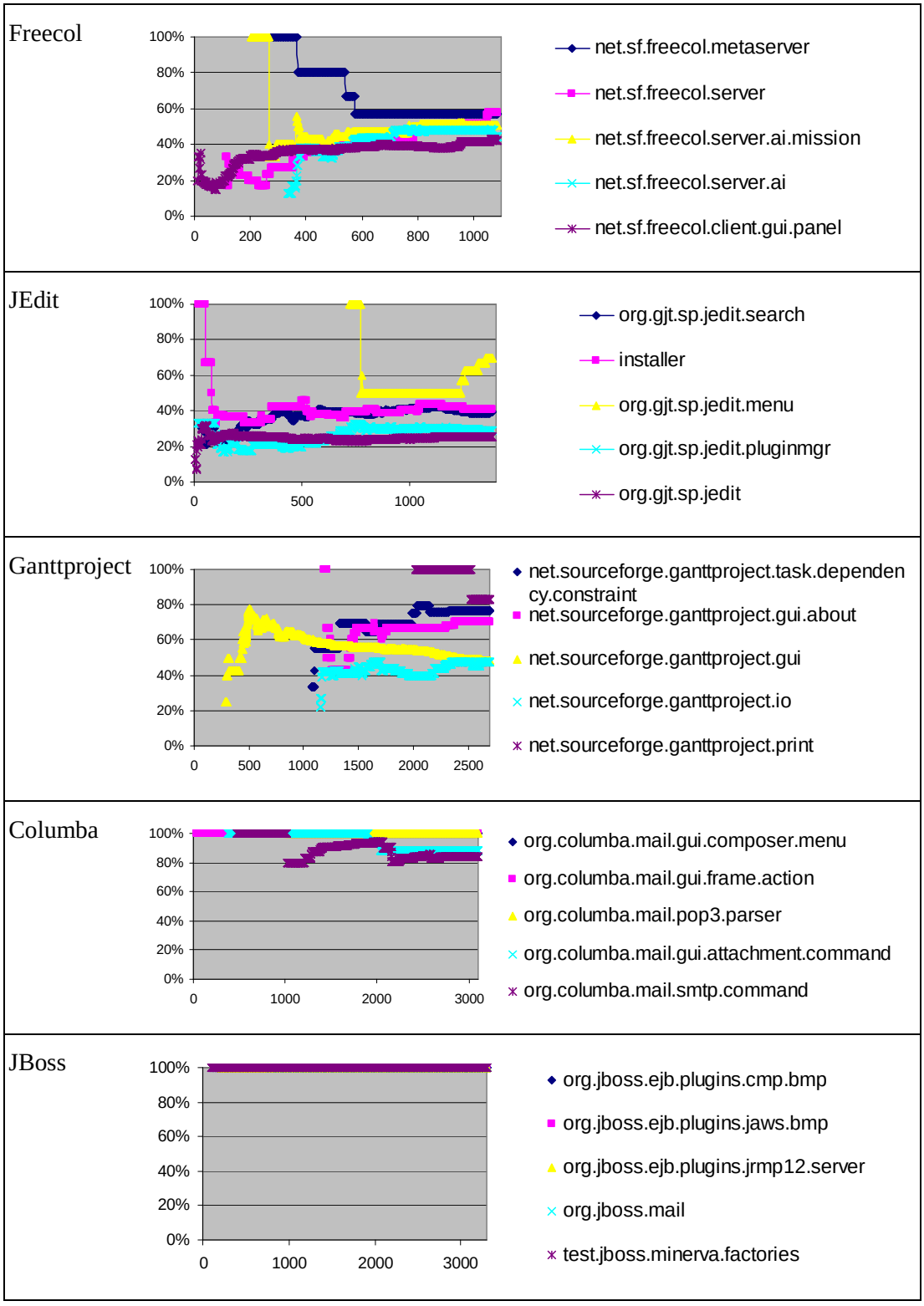


Figure 9-14. Evolution of instability of packages with the highest instability due to methods cloned.

We also analyze the percentage of changes in cloned methods per package, i.e. applying the formula ***stability_contribution*** to the packages of the applications analyzed. That is, from all the changes of methods inside a package, how many of these changes occurred when the methods were cloned. The idea is to calculate the accumulated percentage of change related with cloned methods after each change in the application. The results are shown in Figure 9-14, which depicts a curve per package analyzed. On the x-axis, it has the commit transactions in the application, and on the y-axis, it has from all the changes in the methods of a package which percentage occurs in methods cloned.

According to results (Figure 9-14), some of the packages with highest percentage of cloning in the application have also high instability when cloned, e.g. `net.sf.freecol.client.gui.panel`, `org.gjt.sp.jedit.search`, `org.gjt.sp.jedit`, `net.sourceforge.ganttproject.gui`, `net.sourceforge.ganttproject.io` (see Table 9-1). This result was expected given that most of the methods in these packages were cloned methods; therefore, most of the changes in these packages would happen inside cloned methods.

The packages whose instability is close to 100% are packages with a few methods, most of them cloned, that almost never changed. Therefore, a single change of a cloned method could cause instability of 100% for the package's lifetime. This is a common problem of using percentages when the number of events analyzed is small.

However, packages with a typical size and percentage of cloning are more susceptible to be changed in cloned methods. Table 9-2 shows that cloned methods are more likely to change than methods without clones. Note that the percentage of clone methods that changed is larger than the percentage of methods not cloned that change in every application, and that the difference is considerably larger for cloned methods.

Table 9-2. Average percentage of methods that change, cloned vs. not cloned.

	Methods with clones that change	Methods without clones that change	Increase of chance of changing for cloned methods
Freecol	65%	31%	109%
JEdit	65%	44%	47%
Ganttproject	39%	11%	254%
Columba	38%	22%	72%
JBoss	45%	22%	104%

9.2.3.1 Stability in methods

We verify to what extent the changes in cloned methods were indeed inside of the cloned fragment. Hence, we measured the instability inside methods caused by the cloned fragment. The results are summarized in Figure 9-15, which presents the results of the formula ***stability_SCE*** presented in section 9.1.3. Figure 9-15 contains the average stability inside methods cloned for all the commit transactions of the applications analyzed. That is, the x-axis has the commit transactions in sequential order, and the y-axis has the average percentage of all changes inside the cloned fragments while the methods were cloned from all the methods that have been cloned by that commit transaction.

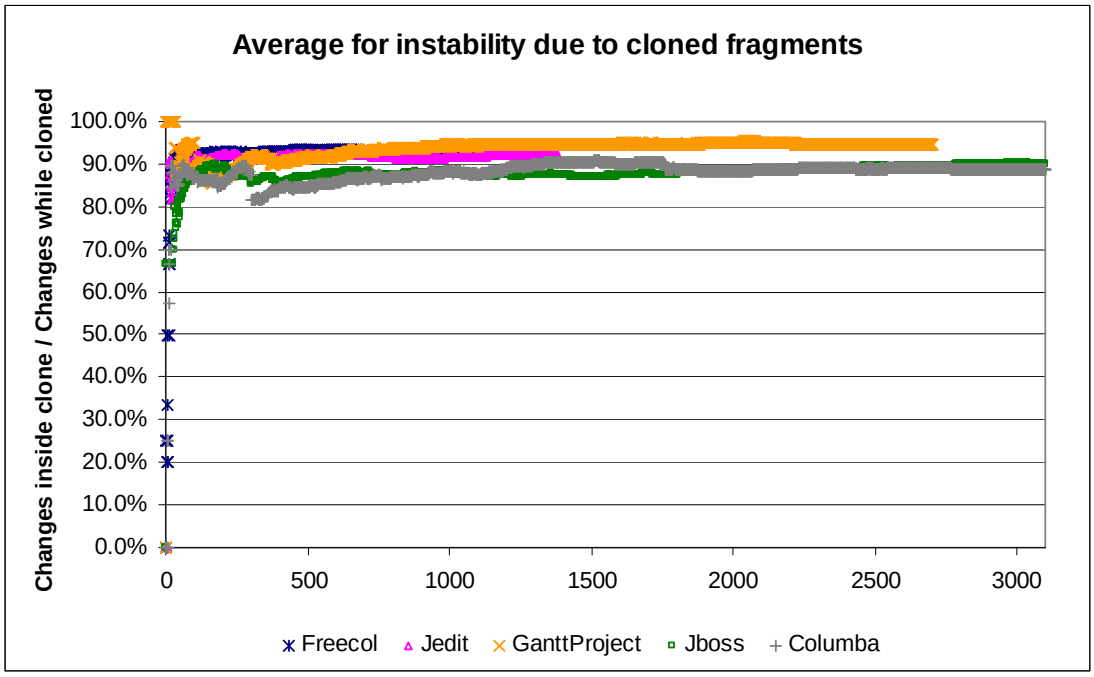


Figure 9-15. Instability due to cloned fragments

We have found that most of the changes in a method when it is cloned occur inside the cloned

fragment, which supports the assumption that methods are an appropriate level of granularity to analyze cloned code. This result is consistent with the one that indicates that cloned fragments tend to cover the totality of the method (see section 8.2.7). However, one should not discard the hypothesis that the extra changes in cloned methods are due to changes in the cloned fragment, given that the families tend to be volatile but the clones inside the methods do not.

By analyzing at the instability of cloned methods we have found that cloned methods are more likely to be changed; therefore, some packages would be highly affected by the extra changes due to cloned methods, and that most of the changes inside cloned methods are changes to the cloned fragment.

9.3 Summary

This chapter tells how to use the data on the evolution of the SCIUS to assess its general effect on an application. We have found that the percentage of cloning in the application tends to remain stable over time, and that it is usually between 10% and 20% of the application. Although there is not much information in the progression of the characteristics over time, the graphs helped to identify critical moments regarding the cloning in the application, i.e. when several clones have been added, or deleted. These key events have been found whenever external libraries are integrated, and when the code is restructured.

We have found also that the percentage of cloning might be related with the modularity of the application. There are two reasons for this affirmation. First, packages with high number of methods also had a high percentage of cloned methods. Second, the applications with a more even distribution of responsibilities, tended to have also a more even distribution of clones.

We were expecting to find that packages cloned were those in charge of repetitive code such as those that handle APIs or those that customize functionalities for different platforms. Although the packages handling IO and GUI showed a high percentage of cloning, packages handling the main functionality of the application were also highly cloned.

The extension of cloning inside methods (i.e. the clone coverage) is reduced over time. The rate of this reduction could be explained by the degree of similarity of methods in the family. Exact clones might be more likely of maintain the similarity, and therefore, their clone relations. Cloned fragments of families that differ more may be more likely to be changed inconsistently, and to be fragmented for customizing the functionality cloned.

We have confirmed the finding of the previous chapter that indicates that cloned methods tend to remain cloned all their lifetime.

That there are three types of event that may alter the evolution of the persistence of clones in an application: additions of cloned methods, removal of clones from methods that are not deleted, and introduction of clones to existing methods.

Finally, the stability analysis shows that cloned methods change more than methods not cloned. Moreover, among the packages that have higher instability are those packages that have the highest levels of cloning. Finally, the stability analysis also shows that the majority of changes inside cloned methods occurred inside the cloned fragment, which is consistent with the fact that most of the clones cover the majority of the method.

The analysis of the extension permits to identify the magnitude of the SCI in the application, and at the level of the SCE. This is a first approach to assess its potential harmfulness. The evolution of the extension permits to evaluate if the SCE affected at the beginning are the only ones that are affected in the application, or if the SCIUS expands to other SCEs over time. Finally analyzing which packages are more susceptible to have the SCIUS may help to identify which concerns are related with the SCIUS.

The analysis of the persistence permits to know to if the SCIUS is a long-term issue. If the SCIUS tends to disappear rapidly, there is a chance that it does not affect the maintainability of the application. The contrary occurs if the SCIUS is persistent. Analyzing which packages may have volatile SCIUS could help to identify harmless SCIUS.

The analysis of the stability of the SCEs with the SCIUS in comparison with those that do not have it reveals whether or not the SCIUS may affect the SCE that hosts them. In addition, the analysis of stability inside the SCE permits to assess accurately to what extent the stability measured in the SCEs with the SCIUS is indeed due to the SCIUS. However, the analysis of stability over time and per package does not reveal much information.

The next chapter explains how to establish if having the SCIUS affects the changeability of SCEs that hosts them.

Chapter 10. Effect of the SCIUS on changeability

The effect of SCIUS on the changeability of SCEs is a phase achieved through four sub-phases (see Figure 10-1). This chapter explains a proposal to evidence the effect of SCIUS on changeability: From finding whether the SCIUS has any effect or not to predicting the effect of the SCIUS on changeability based on thresholds for SCEs or SCIUS characteristics. The chapter also presents a case study that analyzes the effect of clones in the changeability of methods.

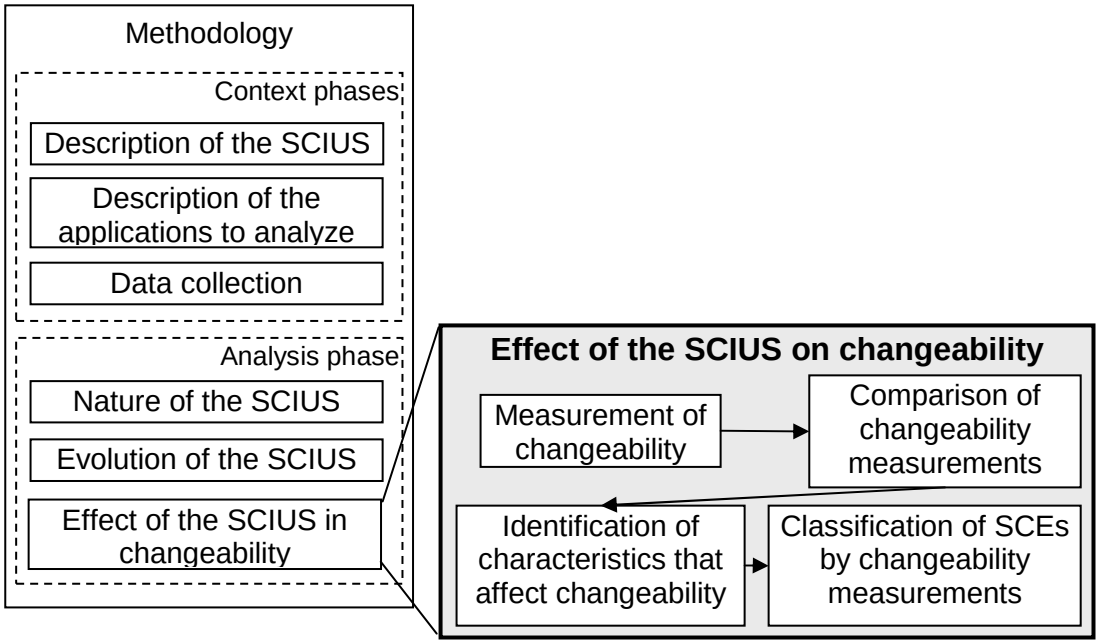


Figure 10-1. Description of the phase "Effect of the SCIUS on changeability".

Note that, the effect of the SCIUS on the changeability of the SCE can be magnified or reduced depending on the results on the evolution of the SCIUS (previous phase); in this way, combining the results from this phase and the previous phase would produce an overall assessment of the effect of the SCIUS in the application. It is unlikely that the results on stability would be contrary to the results on changeability. However, results on persistence or extension alone may lead to different results to those obtained in changeability. If the SCIUS has no effect on the changeability of SCEs, it would be a harmless SCIUS in terms of

changeability. If the SCIUS affects the changeability of the SCEs that hosts them, the SCIUS is harmful. Nevertheless, the overall harmfulness of the SCIUS changes depending on its persistence and extension. If the SCIUS has very low persistence and extension, it would be a harmful SCI but its effect would be ephemeral and limited to few SCEs. If the SCIUS has low extension and high persistence, it would be a harmful SCI persistent but limited to few SCEs. If the SCIUS has high extension and low persistence, it would be a harmful SCI affecting a large percentage of the SCEs but volatile. Finally, if the SCIUS has low extension and high persistence, it would be a very harmful SCI affecting a large percentage of the SCEs for a large percentage of their lifetimes.

Intermediate results on the analysis of clones at the level of methods using some steps proposed in this chapter have been published in [Lozano '07a; Lozano '08b; Lozano '08c]

10.1 Phase description

This section explains the goals of each sub-phase of analyzing the effect of the SCIUS on changeability, and the steps required to accomplish each sub-phase. The first sub-phase, called measurement of changeability, presents the metrics proposed to assess the changeability of a SCE in a given *period*. Periods permit analyzing the effect of any source code characteristic of SCEs because it is possible to compare the periods in which a SCE has and does not have the characteristic. In particular, the metrics proposed are required to perform the rest of the analysis of the effect of the SCIUS in the changeability of a SCE. However, note that the concept of periods can be used to propose metrics for evaluating maintainability attributes other than changeability.

The second sub-phase, called comparison of changeability measurements, explains how to assess whether or not the SCIUS has an impact in changeability or not. Moreover, it explains how to check if the SCIUS has a positive or negative effect.

The third sub-phase, called identification of characteristics that affect changeability, proposes some tests to identify characteristics on the SCE or on the SCIUS that may aggravate the impact of the SCIUS on changeability.

The fourth and final sub-phase, called classification of SCEs by changeability measurements, aims to identify the combination of characteristics of SCEs and SCIUS that increase the effect of the SCIUS in the changeability of the SCE.

Each sub-phase is presented in the following sub sections.

Deliverable: An analysis on the effects that SCIUS has on changeability of SCEs.

Rationale: This step aims to measure whether or not having SCIUS in the SCEs increases the changeability decay i.e., if having the SCIUS increases the difficulty to change those SCEs. If the SCIUS indeed affects the changeability of SCEs of the application, its removal should be a priority, especially if it is persistent problem, and even more if it affects a significant fraction of the SCEs (i.e. extension).

Procedure: First, assess if the changeability of SCE is different when having the SCIUS, by comparing changeability decay metrics (see section 10.1.1).

In case the changeability differs when there are SCIUS (section 0), measure how different it is by calculating the increase in changeability metrics when SCEs have SCIUS.

Regardless of the effect of the SCIUS on changeability, identify the attributes of SCEs that seem to affect the changeability metrics by correlating the value of the attribute and the value of changeability metrics (section 10.1.3). In case there is a difference on the changeability of SCEs when having the SCIUS, identify which attributes of the SCIUS seem to affect the changeability metrics by correlating the value of these attributes and the value of changeability metrics.

Finally group the SCEs by their changeability metrics and find if there are attributes of the SCEs or of the SCIUS that characterize each group (section 10.1.4).

10.1.1 Measurement of changeability

Deliverable: Each SCE should have one or three sets of changeability values, depending on the persistence of the SCIUS inside it, as shown in Figure 10-2. If the SCE never have SCIUS, the SCE belongs to a set of SCEs called **Never with a source code Issue**; this set is identified by the acronym **NI**. If the SCE have SCIUS all its lifetime, the SCE belongs to a set of SCEs called **Always with a source code Issue**; this set is identified by the acronym **AI**. If the SCE have SCIUS for a fraction of its lifetime, the SCE belongs to a set of SCEs called **Sometimes with a source code Issue**, this set is identified by the acronym **SI**. SCEs AI and NI only have one set of changeability values calculated over their entire lifetime. SCEs SI have three sets of changeability values calculated: over their entire lifetime, over the logical changes in which they have a SCIUS, and over the logical changes in which they do not have a

SCIUS.

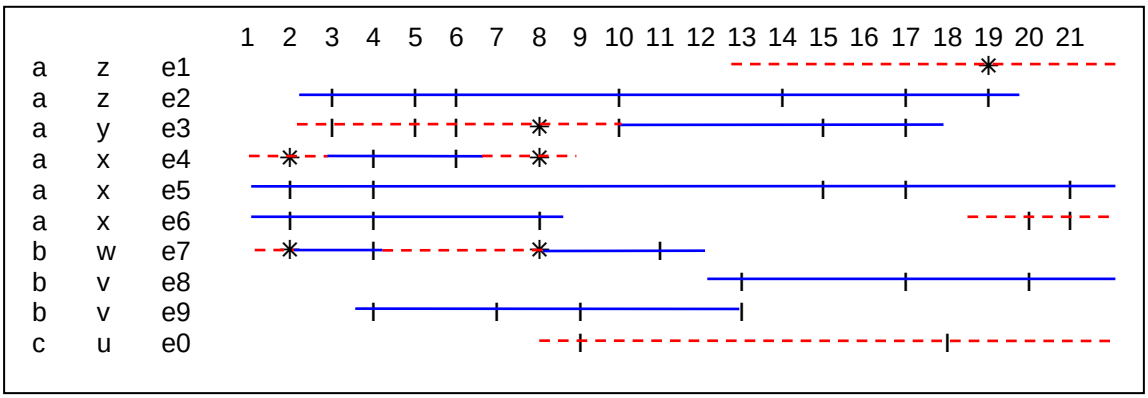


Figure 10-2. SCEs AI, NI, and SI. Periods with the SCIUS are red dashed lines. Periods without the SCIUS are blue straight lines. SCEs AI are `a.z.e1`, and `c.u.e0`. SCEs NI are `a.z.e2`, `a.x.e5`, `a.x.e6`, `b.v.e8` and `b.v.e9`. SCEs SI are `a.y.e3`, `a.x.e4`, and `b.w.e7`.

Rationale: It is necessary to have different sets of changeability values depending on the persistence of the SCIUS on the SCE to be precise with the comparison of measurements. To be accurate with the effect of SCIUS on the changeability of SCEs, it is necessary to calculate the changeability with respect to the periods in which there were SCIUS in the SCEs. In this way, it is possible to compare the changeability of SCEs SI when they have the SCIUS vs. SCEs SI when they do not have the SCIUS. Moreover, it is possible to calculate the increase or decrease of changeability due to the SCIUS. Although the changeability change cannot be calculated for SCEs AI or SCEs NI because there is no baseline changeability to compare with, it can be calculated for SCEs SI because one can compare the changeability of the same SCE with and without the SCIUS. Another advantage of such separation is that having the changeability for all SCEs, for all their lifetime permits to analyze to what extent the fact of having a SCIUS can be associated to a range of changeability values. Finally, it is possible to evaluate to what extent the changeability values depend on characteristics of the SCE or of the SCIUS.

Procedure: Eliminate atypical logical changes i.e. those that are in the top of the list of logical changes when they are ordered by the percentage of SCEs changed. The changes eliminated should be in the top 5% of the list. Atypical changes may create noise in the analysis because all the SCEs modified in atypical changes would have an artificial increase of their changeability.

Classify the SCEs among the three sets defined above (i.e. AI, NI, SI). For all the SCEs calculate their changeability decay for all their lifetime. For those SCEs belonging to SI, calculate their changeability decay for their periods with the SCIUS, and for their periods

without the SCIUS. A **period** is a subset of the logical changes, i.e. the set of periods is ***P=set(LC)***. The logical changes composing a period are not necessarily sequential, which permits to reason about SCEs that are intermittently with the SCIUS.

It is necessary to ensure that the periods compared have a comparable interval, so they have the same likelihood of presenting changes. Note that if one of the periods is very small, its chances of presenting changes are smaller, and any change could have a significant impact. In order to have comparable periods for SI-methods, we suggest to eliminate from the set of SI those SCE whose lifetime with the SCIUS is less than the 15% of the lifetime of the SCE. This percentage is suggested from manual analysis on different sets of methods.

The changeability decay is calculated as a set of values for four metrics that measure the effort required to change a SCE on a set of logical changes. Below there are details on the measurements introduced to assess changeability decay.

The effort of changing a SCE may increase if the SCE requires more changes. To assess if the effort increases because of the amount of changes there are two metrics: likelihood and frequency. The effort of changing a SCE may also increase if the changes to that SCE are more complex. To assess if the effort increases because of the complexity of the changes there are two metrics: impact and depth. The value of the measurements ranges between zero and one. The higher the value of the measurement, the more difficult the change, and therefore, the measurement indicates the loss of changeability, or changeability decay.

The history of the SCEs shown in Figure 10-2 is used to exemplify the calculation of the measurements. Suppose that they represent the whole history and the whole set of SCEs of a fictitious application.

10.1.1.1 Likelihood

Likelihood indicates whether a SCE changes more than other SCEs. The likelihood is defined as the number of changes to the SCE ***e*** in the period ***p*** (***changes_entity***), over, the number of changes in the period ***p*** (***changes_period***) [Belle '04].

$$\mathbf{likelihood} : \mathbf{E} \times \mathbf{P} \rightarrow \Re$$
$$\mathbf{likelihood}(e, p) = \frac{\mathbf{changes\_entity}(e, p)}{\mathbf{changes\_period}(p)}$$

Equation 10-1. Likelihood

The number of changes a SCE ***e*** in a period ***p*** (***changes_entity***) is defined as the number of times in which ***e*** changed in any of the commit transaction ***ct*** that composed the period ***p***.

The number of changes in the period ***p*** (***changes_period***) is defined as the number of SCEs that changed in commit transactions ***ct*** that composed the period ***p***.

The likelihood can be interpreted as the rate of change required in a SCE ***e*** with respect to other SCEs in the application. For instance, the likelihood of the SCE `a.z.e1` for its period with the SCIUS is `0.06` or `1/15`. The period analyzed goes from commit transaction 13 to 21. In that period, 15 SCEs changed (see the squares in Figure 10-3), while the entity analyzed (`a.z.e1`) only changed once (see the circle in Figure 10-3).

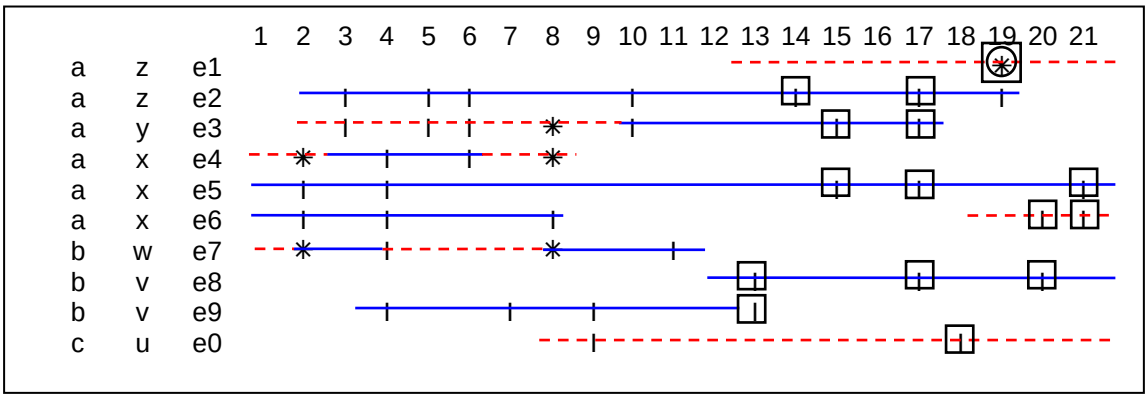


Figure 10-3. Likelihood of SCE `a.z.e1` in the period with the SCIUS. Changes taken into account for the numerator of the formula are marked with a circle. Changes taken into account for the denominator of the formula are marked with a square.

10.1.1.2 Frequency

`Frequency` says how often the SCE changes. The frequency is defined as changes to the SCE ***e*** in the period ***p*** (***changes_entity***), over, the number of commit transactions in that period (***p***).

$$\mathbf{frequency} : \mathbf{E} \times \mathbf{P} \rightarrow \Re$$
$$\mathbf{frequency}(\mathbf{e}, \mathbf{p}) = \frac{\mathbf{changes\_entity}(\mathbf{e}, \mathbf{p})}{\|\mathbf{p}\|}$$

Equation 10-2. Frequency

The frequency can be interpreted as the rate of change required in a SCE to be kept as part of the application. For instance, the frequency of the SCE `a.z.e1` for its period with SCIUS is `0.11` (or `1/9`). The period analyzed goes from commit transaction 13 to 21. In that period, there are nine commit transactions (see the squares on Figure 10-4), in which `a.z.e1` changed once (see the circle on Figure 10-4).

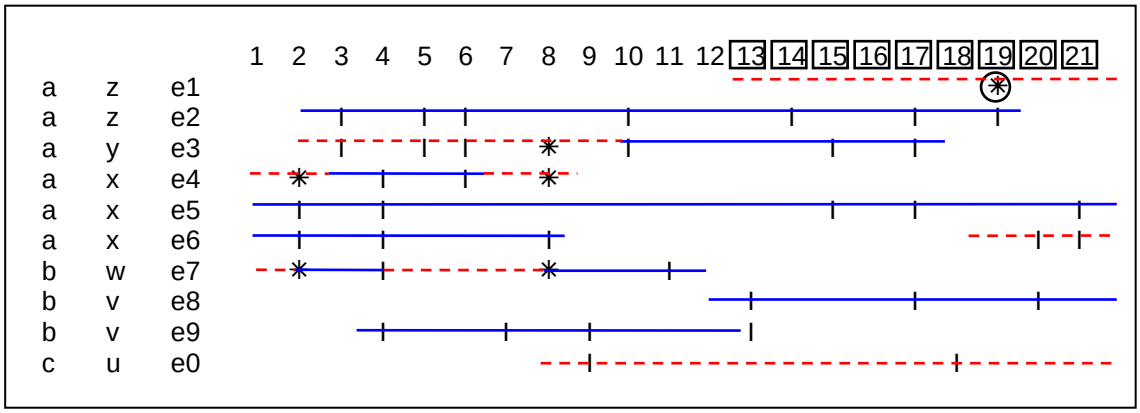


Figure 10-4. Frequency of the SCE `a.z.e1` in the period with the SCIUS. Changes taken into account for the numerator of the formula are marked with a circle. Logical changes taken into account for the denominator of the formula are marked with a square.

Note that a high frequency does not imply that the entity is more difficult to change, e.g. an entity may change frequently because it implements functionality that requires frequent updates. The frequency may indicate the level of difficulty to change an entity, but only in some cases, a high frequency would indicate a high effort. For instance, consider a complex class; implementing a change request may require several physical changes because every time the change request is attempted something else is not taken into account, and the change request remains incomplete or incorrectly implemented. Another example would be a simple class that has a higher frequency of change because it is easier to modify than other classes. Therefore, frequency cannot be considered a changeability decay indicator in isolation. It is necessary to check if other metrics also indicate changeability decay; in such a case, frequency would indicate a higher decay. If that is not the case, frequency would indicate a higher changeability instead of changeability decay.

Finally, notice that cloning would increase the frequency because the chance of being changed would be the number of clone families to which the entity is related times the average size of those families.

10.1.1.3 Impact

Impact indicates the size of the ripple effect of changes that modify a SCE. Impact is the average number of SCEs that co-change with the SCE ***e*** in the period ***p***, over, the number of changes in the period ***p*** [Belle '04].

$$impact : E \times P \rightarrow \Re$$

$$impact(e, p) = \frac{avg\_co\_changes(e, p)}{changes\_entity(e, p)}$$

Equation 10-3. Impact

The average number of SCEs co-changes with the SCE ***e*** in the period ***p*** (i.e. ***avg_co_changes***), is defined as the number of SCEs that co-change with the ***e*** during the period ***p***, over, the number of SCEs in the period ***p***.

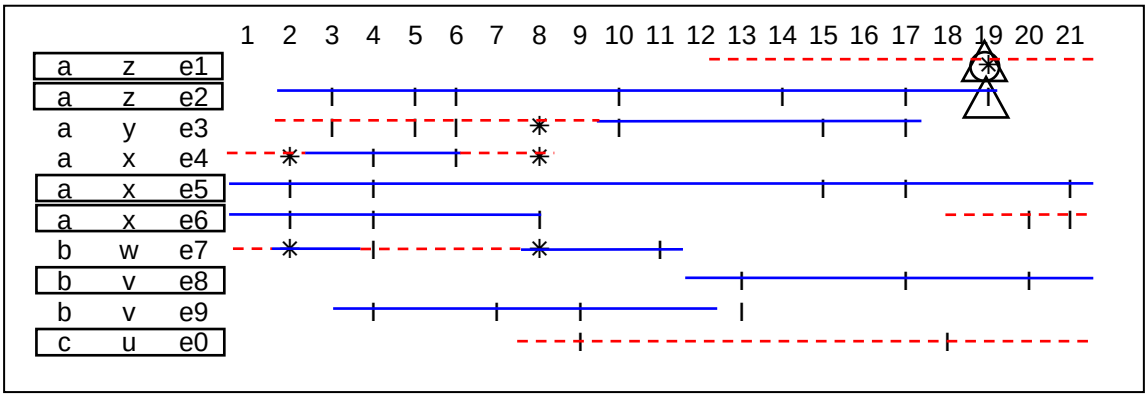


Figure 10-5. Impact of SCE `a.z.e1` in the period with the SCIUS. Co-changes with `a.z.e1` are marked with a triangle; the SCEs that changed in the period are marked with a rectangle. The logical changes taken into account for the denominator of the formula are marked with a circle.

The impact can be interpreted as the scope of changes across the application to keep the application consistent whenever a SCE changes. For instance, the impact of the SCE `a.z.e1` for its period with the SCIUS, is `0.33` (or `(2/6)/1`). The SCE `a.z.e1` only changed once during its period with the SCIUS, which is on the commit transaction 19. Therefore, the denominator of the impact is one (see the circle in Figure 10-5). The average co-changes is `2/6`

because two entities changed in the commit transaction 19 (shown with triangles in Figure 10-5), and six entities composed the application in that commit transaction (shown with triangles in Figure 10-5).

10.1.1.4 Depth

Depth indicates to what extent the structure of the application hides changes. According to the information hiding principle [Parnas '72], source code abstractions (i.e. SCEs) should hide changes. Depth is a metric created to assess to what extent the commits in which a method changes are hidden behind an abstraction. The metric is called depth because if all changes are concentrated in a SCE of low level, for instance, in a method, it means that the change was made deep in the application, and that the change was oblivious to higher SCEs. In this way, the deeper a change is, the easier it is to implement.

For each logical change in the period p, it is possible to identify a SCE that contains all the SCEs of the level of granularity analyzed that were modified. That SCE is of a higher granularity than the one analyzed, and will be called the closest common ancestor of the SCEs modified by the logical change. Depth is the percentage of that SCE (i.e. closest common ancestor) of that was not changed. The depth is also the inverse of the average density of the changes of the SCE e in the period p.

$$depth : E \times P \rightarrow \Re$$
$$depth(e, p) = 1 - avgDensity(e, p)$$

Equation 10-4. Depth

The average density (**avgDensity**) indicates which percentage of the closest common ancestor of the SCEs changed in p whenever e was changed. This means, the number of SCEs that changed on each commit transaction in the period p that modified e, over the number of SCEs that composed the closest common ancestor of the SCEs changed on each commit transaction in the period p that modified e.

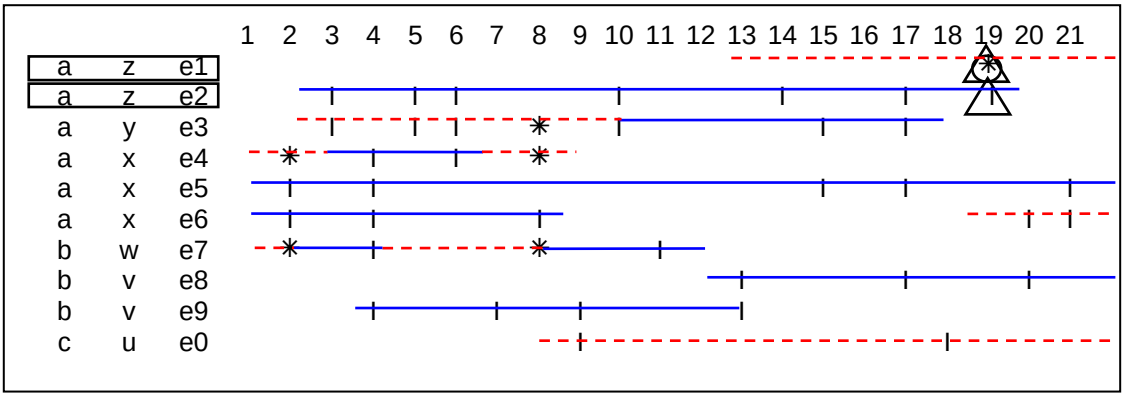


Figure 10-6. Depth of the SCE `a.z.e1` in the period with the SCIUS. Physical changes that composed logical changes in which the SCE was modified are marked with a triangle, SCEs considered for identifying the common closest ancestor are marked with a rectangle. Changes to the SCE are marked with a circle.

The depth can be interpreted as the distance among the SCEs changed whenever a given SCE changed. For instance, see in Figure 10-6 the depth of the SCE `a.z.e1` for its period with the SCIUS, that is `0` (or `1 - ((2/2)/1))`. The `(2/2)` are obtained from the two SCEs changed in the 19th logical change, over the two entities that composed their closest common ancestor `a.z` in that logical change. The `(/1)` part comes from the only commit transaction in which the SCE `a.z.e1` changed during the period analyzed. Given that all the SCEs were changed were part of the same SCE of higher granularity (`a.z`), the depth is zero. A change with depth zero is ideal because it is confined to the SCEs that compose another SCE, which means that the structure hides and encapsulates the complexity of the change.

Notice that the depth does not necessarily measure encapsulation because the minimal common ancestor of those entities changed in a commit transaction may not encapsulate them in an abstraction of the same level. Consider for instance that the entities on Figure 10-6 are methods; therefore, the entities in the figure should be interpreted as follows: the method `e1` belongs to the class `z,` and the class `z` belongs to the package `a`. Notice that in commit `3` only methods `e2` and `e3` change ;their minimal common ancestor would be package `a`. In commit transaction `3` the package `a` only has `5` methods i.e. `e2`, `e3`, `e4`, `e5`, `e6`. This means that the depth of changes in the commit transaction `3` is `0.6` (or `1 - ((2/5)/1)`). This result can be interpreted as package `a` did not hide the changes occurred in commit transaction `3` because 60% of its methods were not modified. Now consider the commit transaction 4. The methods `e4`, `e5`, `e6`, `e7`, and `e9` changed; but they do not have a common ancestor, so, the assumed ancestor is the root of the application. Apart from those methods that changed in the commit

transaction 4, the other methods that are part of the application are `e2` and `e3`. Therefore, the depth of changes in the commit transaction `4` is `0.28` (or `1 – ((5/7)/1)`); which would indicate that the abstraction hide the changes. So, the metric shows the percentage of the class, package, or application that is not modified when calculated for a commit transaction. When calculated for a period, the metric shows the average percentage of the entities that were not hidden by an abstraction.

10.1.2 Comparison of changeability measurements

To measuring the effect of SCIUS on the changeability of the SCE where it is placed, it is necessary to answer two questions: first if the SCIUS has an effect at all, and second if the effect of the SCIUS can be considered positive or negative. These two questions form the steps to achieve the comparison of changeability measurements as Figure 10-7 shows. Each step is described in detail below.

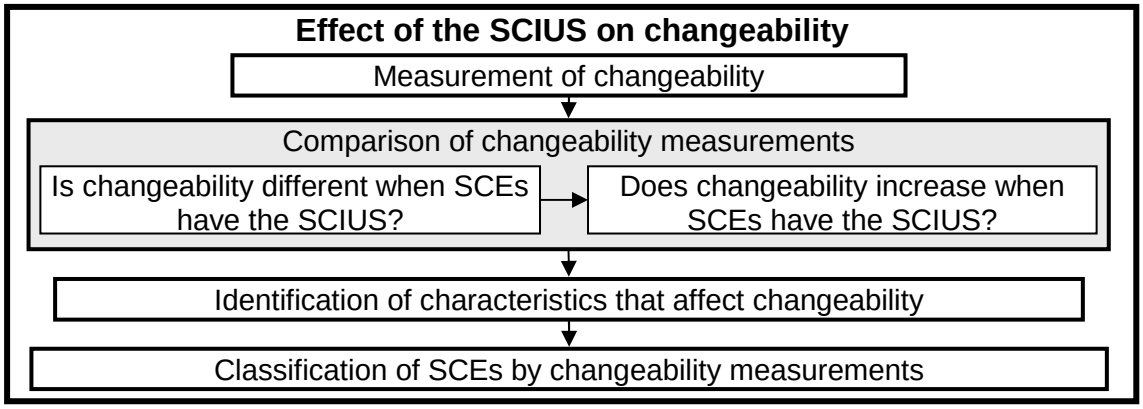


Figure 10-7. Description of the sub-phase "Comparison of changeability measurements" of the phase "Effect of the SCIUS on changeability".

10.1.2.1 Is changeability different when SCEs have the SCI?

This steps permits to say with certainty that the SCIUS affect the changeability of the SCEs where it occurs.

Deliverable: A statistical validation of the difference on changeability decay of SCEs when having the SCIUS.

Rationale: SCIs are believed to be harmful. Therefore, the natural hypothesis is that they increase the changeability decay of the SCEs that have them. To support this hypothesis it is

necessary to establish that SCEs without SCIUS have different changeability decay from those SCEs with them.

Procedure: To find if there is a difference on changeability when SCEs have the SCIUS, it is enough to compare statistically the distributions of the changeability decay metrics for the SCEs when they have the SCIUS, and when they do not. However, it is necessary to choose the appropriate statistical test. The appropriateness of a statistical test depends on the properties of the distributions (see Appendix B section B.2.1). We propose to compare the distribution of each changeability metric for the SCEs AI vs. NI, and for the SCEs SI for their periods with the SCIUS and without the SCIUS, for each case study. Given that the changeability of SCEs SI are compared against themselves with and without the SCIUS, their comparison must be *paired*. On the contrary, given that the changeability of SCEs AI is compared against the changeability of other SCEs (those NI), the test cannot be *paired*.

The p-value obtained from the tests indicates if the changeability metric compared is different or not for SCEs with different SCIUS-circumstances (i.e. AI vs. NI, SI with SCIUS vs. SI without SCIUS). If the value is below 0.05 it means that the distributions are different, if the p-value obtained is above 0.01 it is impossible to state statistically that the distributions compared are different.

10.1.2.2 Does changeability increase when SCE have the SCI?

In case the previous step of the phase finds that the changeability changes for SCEs with the SCIUS, it is still necessary to establish if the changeability decays or improves. To assess if the effect of the SCIUS on changeability is positive or negative we propose three tests, both aiming to assess the difference of changeability decay with the SCIUS and without it . The first describes the distributions to compare their average behavior. The second test plots the distributions to establish which one has the higher values. The third test plots the difference of changeability metrics for SCEs SI.

→ *Comparison of quartiles*

This test aims to establish key values in the distributions to find which distribution tends to have higher values.

Deliverable: A comparison of the quartiles of each distribution of changeability metrics. That means four numbers for each distribution (i.e. for each application of study, for each metric, for

the each set of SCEs, for each period in the set).

Rationale: Quartiles indicate the values of every 25% of the distribution. If the quartiles of a distribution are higher than the quartiles of other distribution, then the values of the first distribution tend to be greater. Given that the distributions have changeability metrics for periods with and without the SCIUS, comparing their quartiles may indicate if the changeability of SCEs is much higher with the SCIUS.

Procedure: For each application to analyze, and for each changeability metric, compare the quartiles of the distributions AI vs. NI, and SI with the SCIUS vs. SI without the SCIUS. The quartiles of a distribution are values that divide the distribution into four equal parts. Quartiles are obtained by organizing the values of the distribution, and taking the value every 25% of the distribution (see Appendix B section B.1.1). If for all applications to analyze, if the values of the quartiles of a changeability metric tend to be greater for the distribution with the SCIUS (i.e. AI, or SI with the SCIUS), it is likely that the SCIUS has a negative effect on that changeability metric.

→ *Comparison of distribution graphs*

Comparing the distributions by quartiles is a useful test only if the distributions are very different. For more subtle differences, it is better to plot the distributions themselves.

Deliverable: A graph comparing the distribution of changeability metrics for the periods with and without the SCIUS. For each changeability metric, and for each case study, a graph with the curves of the distribution for the periods with the SCIUS, and of the distribution for the periods without the SCIUS.

Rationale: Plotting the distributions provides a clear image of the amount of cases per value. If one distribution is mostly on the right side of the other, it means that its values are typically higher. If both distributions are within the same range of values, the one that has its peaks to the right would be the one with higher values.

Procedure: For each application to analyze, and for each changeability metric, plot the distributions AI vs. NI, and SI with the SCIUS vs. SI without the SCIUS. A distribution plot is a graph that has in the x-axis the different values in the data set organized from the lowest to the greatest from left to right, and in the y-axis the percentage of cases in which that value appears in the data set (see Appendix B section B.1.2).

If for all applications to analyze, for each changeability metric, the curve representing the distribution with the SCIUS (i.e. AI, or SI with the SCIUS) is to the right or its peaks are to the right of the curve representing the distribution without the SCIUS, it is likely that the SCIUS have a negative effect on that changeability metric.

→ *Graphs of increase of changeability metrics when the SCEs have the SCIUS*

This step is the only one that actually measures the difference of changeability with and without the SCIUS.

Deliverable: For each application to analyze, and for each changeability metric: a graph of the distribution of differences of changeability metrics for the SCEs SI for periods with and without the SCIUS.

Rationale: Plotting the distributions of the difference of changeability measurements permits to establish if the SCIUS have mostly a negative or a positive impact on the changeability of the SCEs because they show the differences between periods and how frequently these differences occur. If the peak of the curve is located to the right of zero, it means that in most of the cases the changeability with the SCIUS is higher than the changeability without the SCIUS. The further the peak is from zero, the higher the difference, and therefore the higher the impact is.

Procedure: For each application to analyze, for each changeability metric, and for each SCE SI, from the changeability for the period with the SCIUS take away the changeability for the period without the SCIUS.

For each changeability metric, depict the increase of the metric for SI-entities when they have the SCIUS with respect to the metric when they do not have the SCIUS. To depict the increase calculate for each SC-method and for each metric. The effect is defined as the ratio of the increase or decrease of the metric between periods with respect to the value of the metric during the not cloned period:

$$increase(M, m) = \frac{M(m, P_C(m)) - M(m, P_{NC}(m))}{M(m, P_{NC}(m))}$$

Equation 10-5. Increase of changeability decay metrics when the SCE has the SCIUS

where M is the metric, m the method analyzed, $P_C(m)$ is the period cloned for the method m, and $P_{NC}(m)$ is the period not cloned for the method m.

To plot it, lay in the *y* axis the values for the increase of the metric, and in the *x* axis the

percentage of SI-methods. Place a dot for the percentage of methods that presented at most that metric increase.

10.1.3 Identification of characteristics that affect changeability

We propose two types of tests to find evidence that indicates that characteristics of the SCIUS or of the SCE can be correlated with changeability decay. The two types of tests are: graphical correlation tests, and statistical correlation tests (see Figure 10-8). This section explains how to use these tests to detect characteristics that affect changeability. The following sub-sections explain in detail the tests.

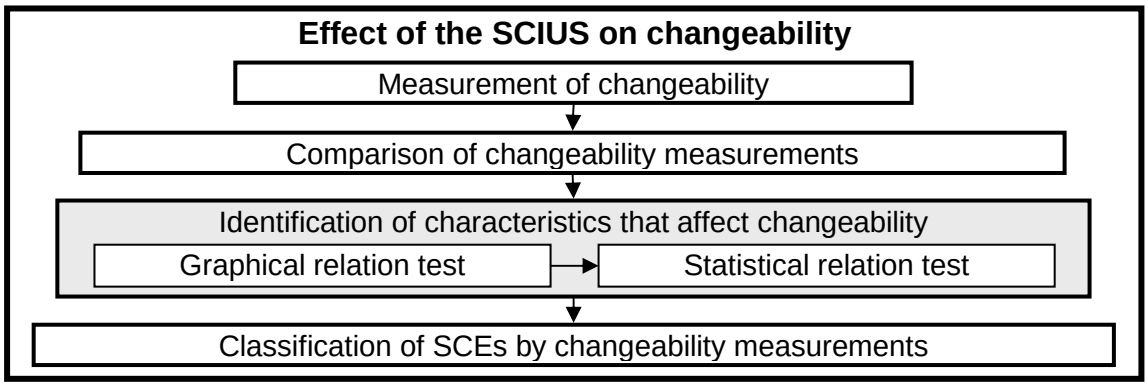


Figure 10-8. Description of the sub-phase "Identification of characteristics that affect changeability" of the phase "Effect of the SCIUS on changeability".

Deliverable: Results of the graphical and statistical relation tests for those characteristics that are related to any of the changeability metrics for all applications to analyze.

Rationale: The goal is to obtain the most specific set of characteristics correlated with the changeability of SCEs. To achieve this goal we propose to test pairs of characteristics of SCEs and SCIUS that combined show a stronger correlation with changeability of the SCEs than the correlation found separately. Therefore, it is necessary to find which characteristics of SCEs affect changeability regardless of the SCIUS. In addition, it is necessary to find which SCIUS characteristics affect the changeability of the SCE where they are placed. The identification of characteristics of SCEs that are correlated with changeability have two advantages: they may indicate SCE characteristics that impact changeability more than the fact of having the SCIUS, and they may be helpful to detect SCIUS that are harmless only when they are in certain SCEs.

Procedure: Identify characteristics that may affect changeability of the SCIUS and of the SCE.

Each characteristic must be compared against the values of each one of the changeability metrics.

For each application to analyze, calculate the average changeability value for the following SCEs, and periods: SI with the SCIUS, SI without the SCIUS, SI all lifetime, AI all lifetime, and NI all lifetime.

For each application to analyze, calculate the average value of the characteristics of the SCEs for the following SCEs, and periods: SI with the SCIUS, SI without the SCIUS, SI all lifetime, AI all lifetime, and NI all lifetime.

For each application to analyze, calculate the average value of the characteristics of the SCIUS for the following SCEs, and periods: SI with the SCIUS, and AI all lifetime.

Find if the characteristics affect changeability or not, using the graphical relation test and the statistical relation test (see details below).

- For all SCEs that changed (AI, NI, and SI), check if any of the SCE characteristics can be related with any changeability metric calculated over the SCEs' lifetime.
- For the SCEs SI, check if any of the SCE characteristics can be related with the difference of any changeability metric between the periods with, and without the SCIUS.
- For the SCEs AI and SI, i.e. those affected by the SCIUS, check if any of the SCIUS characteristics can be related with any changeability metric calculated when the SCE has the SCIUS.
- For the SCEs AI and SI, i.e. those affected by the SCIUS, check if any of the SCE characteristics can be related to any changeability metrics calculated over the period when the SCE has the SCIUS.

10.1.3.1 Graphical relation test

The purpose of this test is to identify visually which characteristics grow or shrink accordingly with the value of the changeability measurements of the SCE. The graphs should plot in the x-axis the value of the characteristic analyzed, and in the y- axis the value of the changeability measurement. The relations can be noticed by locating which areas have most of the points plotted. If all points are located in a vertical or in a horizontal stripe, the characteristic is not correlated with that changeability metric, as Figure 10-9*a* shows. If all points are located in a

diagonal stripe with a positive slope, the characteristic is directly proportional to that changeability metric, as shown in Figure 10-9*b*. If all points are located in a diagonal stripe with a negative slope, the characteristic is inversely proportional to that changeability metric, as Figure 10-9*c* shows.

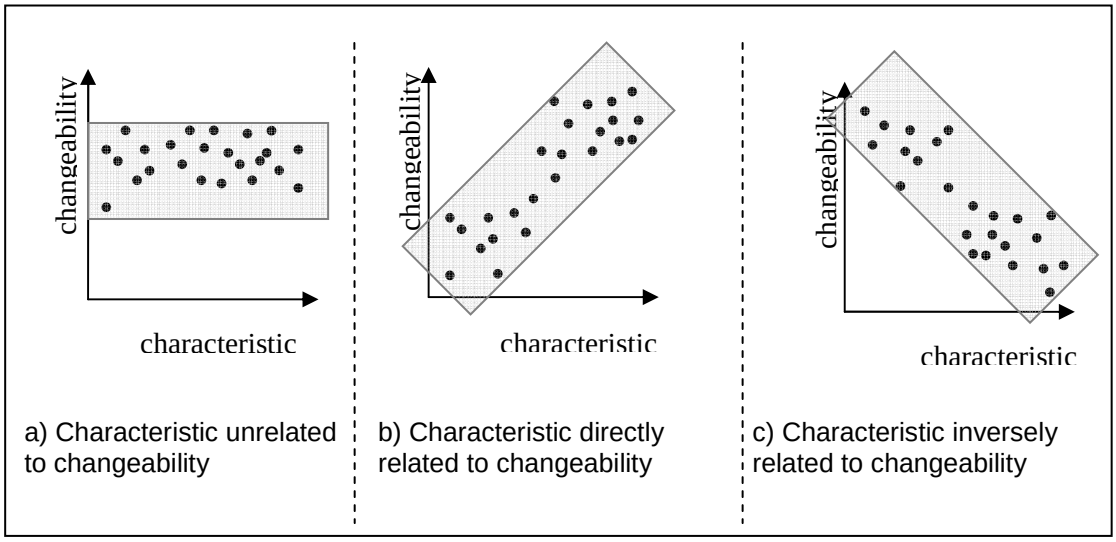


Figure 10-9. Graphical correlation between a numeric characteristic (of the SCIUS or of the SCE) and a changeability measurement.

10.1.3.2 Statistical relation test

We propose to use a metric used in epidemiological studies to assess if being exposed to an agent increases the risk of developing a disease [Green '00], for instance to check if smoking (agent) increases the chances of developing lung cancer (disease). The metric is called relative risk (RR), and is defined as:

$$RR_{ToxicAgent\ Disease} = \frac{\|A \cap D\| / \|D\|}{\|A \cap D'\| / \|D'\|}$$

Equation 10-6. Relative risk

where A corresponds to the set of cases in which the person was exposed to the agent, and D corresponds to the set of cases in which the person developed the disease [Green '00]. The numerator corresponds to the percentage of cases in which the person developed the disease when exposed to the agent. The denominator corresponds to the percentage of cases in which the person did not develop the disease when exposed to the agent. The value of RR indicates the number of times that it is more likely to develop the disease when exposed to the agent. Therefore, a relative risk higher than 1 means that the agent increases the risk of developing the

disease, while a relative risk lower than 1 means that the agent decreases the risk of developing the disease. To avoid false positives, it is recommended to consider only relative risks higher than 2, or lower than 0.5 [Taubes '95]. The higher the relative risk is, the stronger the association is. The fact that an agent is related with the chance of developing a disease does not mean that the agent causes the disease or that other agents have a weaker relation with the development of the disease.

Relative risk can be used as a metaphor for analyzing if source code characteristics increase the risk of developing changeability decay. Changeability decay can be considered a disease because it restricts the chances of the software system to 'survive'. SCIUS can be considered as an agent because it is claimed to be harmful for changeability. Besides, relative risk is useful because it does not require control on the exposure to other agents. Furthermore, characteristics of the SCE and of the SCIUS can be other agents to assess against changeability decay metrics.

Nevertheless, the use of relative risk to analyze the impact of source code characteristics on changeability decay is not straightforward because it is made for boolean values for both the agent and the disease. Therefore, we propose to modify relative risk to explore the strength of the relation between two non boolean characteristics: the agent (A) and the disease (D). The values of the agent characteristics are categorized into three sets: high, low, and normal values. High values are those that form the top 25%, low values are the bottom 25%, and normal values are the rest. The values of the disease characteristics are categorized into two sets: disease and not disease values. Disease values are those that are the top 25%, and not disease values are the rest.

However, categorizing numerical values is not enough to use relative risk. In order to evaluate if the agent is correlated with the disease we consider four cases. Case 1, when a high value in the agent increases the chance of having the disease, i.e. RRHA-D >1. Case 2, when a high value in the agent increases the chance of not having the disease i.e. RRHA-!D>1. Case 3, when a low value in the agent increases the chance of having the disease i.e. RRLA-D>1. Case 4, when a low value in the agent increases the chance of not having the disease RRLA_!D>1. Notice that HA means high value in characteristic A, LA means low value in characteristic A, D means the disease occurs, and !D means it does not. The equations for each one of these cases is shown in Table 10-1.

Table 10-1. Relative risk proposal for non Boolean characteristics.

$RR_{LA\_D} = \frac{LA\ \_\ D / D}{LA\ \_!\ D / !\ D}$	$RR_{HA\_D} = \frac{HA\ \_\ D / D}{HA\ \_!\ D / !\ D}$
$RR_{LA\_!\ D} = \frac{LA\ \_!\ D / !\ D}{LA\ \_\ D / D}$	$RR_{HA\_!\ D} = \frac{HA\ \_!\ D / !\ D}{HA\ \_\ D / D}$

The Table 10-1 illustrates why not all these cases are necessary to evaluate the relation between two characteristics. It is enough calculating RR_{HA_D} and $RR_{LA\_!D}$, *or*, RR_{LA_D} and $RR_{HA\_!D}$ because $RR_{LA\_!D}$ and RR_{LA_D} are inverse one of the other, as well as $RR_{HA\_!D}$ and RR_{HA_D}. We decided to use RR_{HA_D} and $RR_{LA\_!D}$. Intuitively, the meaning of these relative risk formulas is to assess how more dense is the dataset in an area of the graph (see Figure 10-10). A **direct relation** occurs when both characteristics grow or shrink i.e., RR_{HA_D} and $RR_{LA\_!D}$ are greater than one. And an **indirect relation** occurs when one characteristic grows when the other one shrinks, and vice versa i.e., RR_{HA_D} and $RR_{LA\_!D}$ are lower than one, preferably lower than 0.5.

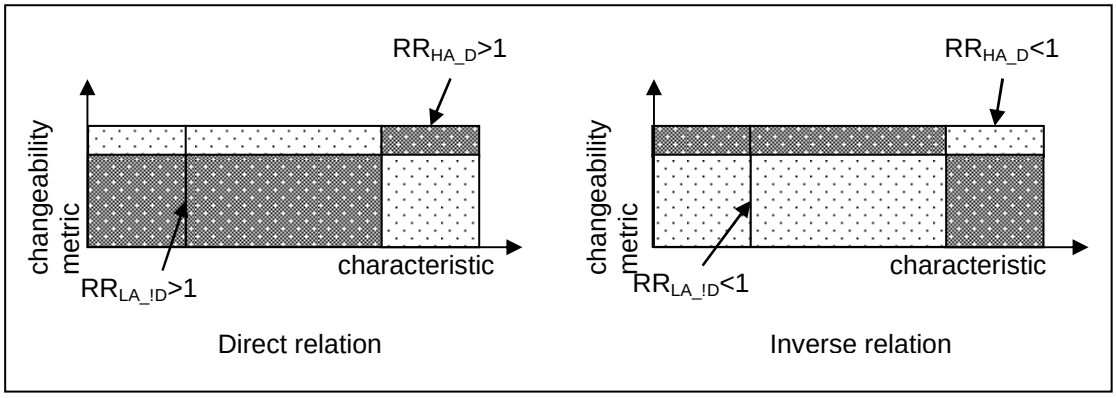


Figure 10-10. Interpretation of direct relation and indirect relation using the adaptation of relative risk proposed

The goal of this part of the analysis is to identify which characteristics might be helpful to predict the effect of the SCIUS in changeability. The agent is a characteristic of the SCE or of the SCIUS, and the disease is a changeability measurement. We expect the characteristics to be directly related with changeability decay i.e. the higher the value of the characteristic is, the higher the changeability decay is. The reason for this is that we expect the characteristic to acts

as Agent (A) that increases the chance of developing changeability decay i.e. the Disease (D). Notice that choosing RR_{HA_D} and $RR_{LA\_!D}$ is convenient because if they have a direct relation, both values would be greater than zero which gives the relation of how many times it is more likely to develop changeability decay (the disease) depending on the value of the characteristic (the agent).

To demonstrate the usage of relative risk in the way proposed consider the following example. Suppose that characteristic A is activity of white globules, and characteristic D is temperature. We would like to know if having fever (high temperature) is an indicator of having an infection (high globule activity). Consider the following data set for 230 cases, of which 54 have fever, 59 have the top 25% white globule activity, and 63 have the bottom 25%:

A = Agent or Characteristic
e.g. activity of white globules

D = Disease or Changeability measurement
e.g. temperature

	LA	MA	HA
D	4	17	33
!D	59	91	26

!LA

!HA

Figure 10-11. Data example to explain the modification of the formula of Relative Risk

From the data above (Figure 10-11), we get the following relative risks:

$$RR_{LA\_!D} = \frac{LA\_!D/!D}{LA\_D/D} = \frac{59/176}{4/54} = 4.5 \qquad RR_{HA\_D} = \frac{HA\_D/D}{HA\_!D/!D} = \frac{33/54}{26/176} = 5.0$$

Given that RR_{HA_D} is greater than one, it means that the risk of having fever is greater when the high white globule activity is high, than when the white globule activity is low. In fact, the risk of having fever is five times greater when the white globule activity is high. Similarly, due to $RR_{LA\_!D}$ being greater than one, whenever the white globule activity is low the likelihood of not having fever is more than four times greater, than when the white globule activity is not low. Therefore, having this example corresponds to the direct case explained above, and we can say that the temperature is an indicator of the white globule activity, because the temperature value

follows the white globule activity.

Notice that it is necessary to check all the two cases of relative risk in order to claim that the characteristics are related. If the data set does not allow concluding either a direct case or an indirect case, it means that characteristic A (agent) cannot be correlated to the characteristic D (disease). Notice also that the value of relative risk gives an estimate of how much the risk of having the characteristic D increases. In order to keep this characteristic of relative risk, we propose to average the relative risks obtained. For example, the relative value of the example shown previously would be 4.75 ([4.5+5]/2). In case an indirect relation is found, it is enough to calculate the average, and invert it so the intuitive interpretation of relative risk is maintained.

10.1.4 Classification of SCEs by changeability measurements

To finish the analysis of the relation between the SCIUS and changeability we propose to identify harmful and harmless instances of the SCIUS using characteristics of the SCE and of the SCIUS. To achieve the identification of SCIUS according to their impact on changeability we propose two steps (shown in Figure 10-12): first grouping SCEs with similar changeability, and second characterizing uniquely each group. Each step is explained in detail below.

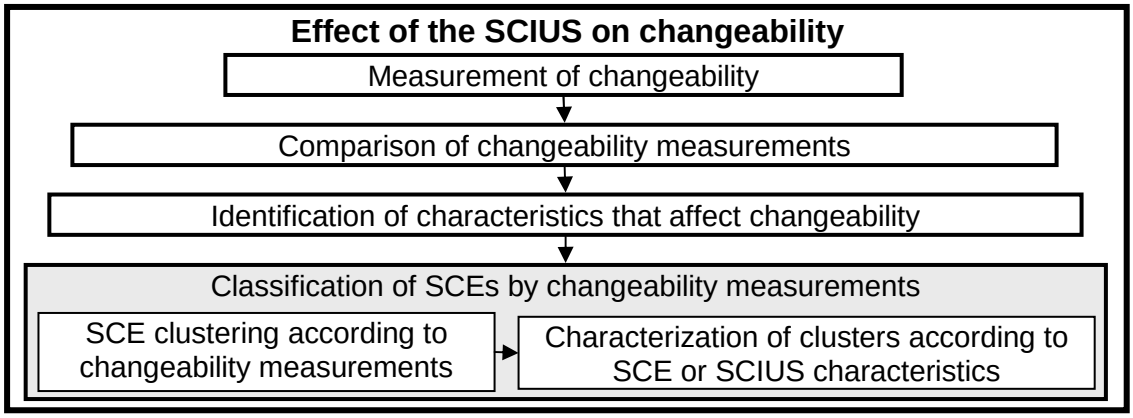


Figure 10-12. Description of the sub-phase "Classification of SCEs by changeability measurements" (in gray) of the phase "Effect of the SCIUS on changeability".

Deliverable: A list of thresholds for SCE or SCIUS characteristics that describe the SCEs whose changeability decay is very high or very low.

Rationale: The purpose of clustering is to identify the 'levels' of changeability that a source code entity can have. The unique characterization of clusters aims to provide patterns to predict which instances of the SCIUS are harmful or harmless.

Procedure: First, merge the data of the SCEs of all case studies into a single dataset. Second, find the groups of SCEs that have similar changeability measurements using any clustering algorithm. Use the changeability measurements calculated for the whole lifetime. Then, describe the changeability of each group. Next, define the interesting groups, which are those with extreme (either very high or very low) changeability values. Afterwards, use classification trees to find the characteristics of the SCEs or of the SCIUS that describe each group. After that, focus on the interesting groups to discover the threshold values of characteristics that are unique of these groups. Finally, corroborate the results with the outputs of the previous step, i.e. checking if the characteristics that uniquely identify the groups with best and worst changeability are those correlated with changeability measurements.

10.1.4.1 SCE clustering according to their changeability

To perform cluster analysis it is necessary to define the similarity for the entities to cluster [Davey '00]. We propose to represent the changeability of each SCE as the vector of its changeability measurements (likelihood, frequency, impact, and depth). The vectors of measurements form a space of four dimensions. The similarity among SCEs would depend on the Euclidean distance of the vectors of changeability measurements. Using this distance, clustering algorithms attempt to discover an inherent structure within the SCEs [Davey '00].

10.1.4.2 Characterization of clusters according to SCE or SCIUS characteristics

Classification trees aim to predict the outcome dataset based on the variables of the dataset. In this case, the outcome to predict would be the set of SCEs with a given changeability level, and the variables are the characteristics of SCEs and or of the SCIUS. The classification trees algorithm recursively divides the data set using the variable that gives the “best” partition of data. The definition of “best” partition depends on the purpose of the analysis. In this case, the best partition is the one by default i.e. the one that differentiates better the data set according to the characteristic to predict. In our case, this means that, the elements of one side of the partition belong to a very different changeability level than the elements of the other side of the partition.

10.2 Phase application

This section comprises the descriptions of the first of the set of analyses proposed on the data gathered.

10.2.1 Measurement of changeability

This section summarizes the calculation of the changeability of methods based on the periods in which they had, or not, a cloned fragment.

Given the results we obtained in previous phases of the methodology, we expect that the percentage of cloned methods to be 20% of the application size (see extension analysis in section 9.2.1). Note that only a fraction of the methods can be analyzed using changeability measurements, i.e. the methods that changed at least once in their lifetime. Given that methods with clones are more likely to change than methods never have had clones (see instability analysis in section 9.2.3), we expect the percentage of cloned methods that can be analyzed to be higher than the percentage of methods never cloned that can be analyzed.

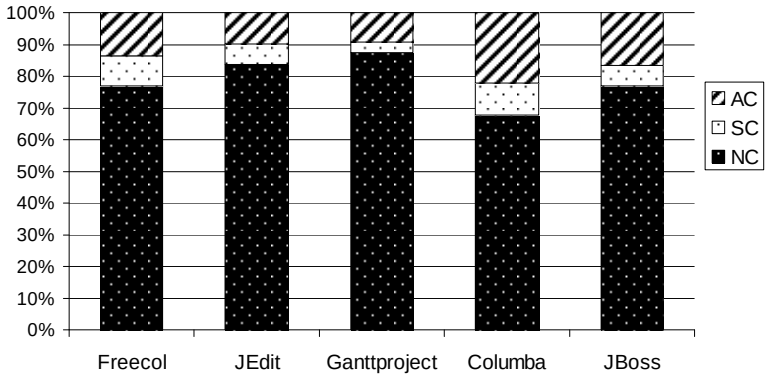


Figure 10-13. Percentage of methods Always Cloned (AC), Never Cloned (NC), and Sometimes Cloned (SC); and from those, percentage of methods changed (i.e. analyzable).

Figure 10-13 summarizes the percentage of methods that were Always Cloned (AC), of methods Never Cloned (NC), and methods Sometimes Cloned (SC). As expected, most of the methods were never cloned in their lifetime. The distribution between AC methods and SC methods is also the same regardless of the application analyzed, i.e. methods with clones only for a faction of their lifetime are the minority of the cloned methods. This means that only a small fraction of each application can be used to measure the difference of changeability before and after being cloned. Table 10-2 summarizes the percentage of methods that changed for each type of methods (AC, NC, SC) for each application. Note that the percentage of methods that can be analyzed is rather small, given that, in order to be able to compare their changeability, these methods should have changes other than their creation. This affects particularly the cloned

methods. In order for an AC-method to be analyzed, it needs to change at least once after its creation, but only 15% to 30% of the AC-methods comply with this requisite. Furthermore, for an SC-method to be analyzed it needs to change at least once on each circumstance: when cloned and when not cloned, the increase in the restrictions has a significant impact on the amount of methods analyzable, as only 5% to 8% of the SC-methods comply with this requisite.

Table 10-2. Percentage of methods analyzable per type of method (AC,NC,SC), from the number of methods analyzable

	Methods changed / No. of methods	AC changed / Methods changed	NC changed / Methods changed	SC changed / Methods changed
Freecol	26%	20%	73%	8%
JEdit	25%	15%	80%	6%
Ganttproj	9%	17%	78%	5%
Columba	10%	30%	64%	6%
JBoss	17%	21%	74%	5%

We present below the median values for each one of the changeability metrics (see Table 10-3 and Table 10-4). The median values of each metric for each application are compared for different types of methods (AC vs. NC). We expect the changeability metrics to be higher for methods that had clones, but also to be higher when the methods have clones in comparison to when they do not have them (when Cloned vs. when not Cloned). Each time the median does not comply with our hypothesis it is highlighted in bold.

Table 10-3. Comparison of the median of changeability metrics for methods Always Cloned (AC) vs. methods Never Cloned (NC)

	Likelihood		Frequency		Impact		Depth	
	AC	NC	AC	NC	AC	NC	AC	NC
Freecol	0.0008	0.0006	0.0046	0.0034	0.0033	0.0035	0.1585	0.1134
JEdit	0.0006	0.0005	0.0038	0.0030	0.0027	0.0025	0.1045	0.1179
Ganttproj	0.0019	0.0009	0.0041	0.0026	0.0016	0.0012	0.2450	0.2093
Columba	0.0008	0.0005	0.0034	0.0024	0.0018	0.0016	0.4208	0.3527
JBoss	0.0008	0.0007	0.0022	0.0018	0.0017	0.0017	0.3333	0.4053

From Table 10-3, it seems that the decay in changeability of methods Always Cloned is higher than the decay in changeability of methods Never Cloned. That difference is even clearer when comparing the changeability metrics when cloned vs. when not cloned (Table 10-4). It seems that the decay in changeability is much higher while a method is cloned (wC), than when a method is not cloned (wnC).

Note that the likelihood, frequency, and impact values for SC-method are usually orders of magnitude higher when cloned than when not cloned. The depth is the only changeability measurement in which the SC-methods have comparable values with the rest of the methods (AC-methods and NC-methods). This indicates that SC-methods are the ones that affect changeability the most, in particular while they are cloned.

Table 10-4. Comparison of changeability metrics for methods Sometimes Cloned (SC) , when Cloned (wC) vs. when not Cloned (wnC)

	Likelihood		Frequency		Impact		Depth	
	wC	wnC	wC	wnC	wC	wnC	wC	wnC
Freecol	0.0094	0.0018	0.0088	0.0018	0.1103	0.0029	0.0907	0.0036
JEdit	0.0067	0.0010	0.0062	0.00093	0.0961	0.0021	0.0981	0.0021
Ganttproj	0.0062	0.0028	0.0075	0.0033	0.1600	0.0025	0.1723	0.0021
Columba	0.0040	0.0009	0.0039	0.0009	0.3425	0.0019	0.3801	0.0017
JBoss	0.0029	0.0011	0.0048	0.0020	0.3897	0.0014	0.3191	0.0014

10.2.2 Comparison of changeability measurements

This section presents the results on the impact of cloning on changeability. Given that cloning is supposed to have several harmful effects for maintaining the application, as discussed in section 5.1.2, we expect cloned methods to have a higher changeability than methods not cloned.

10.2.2.1 Is changeability different when methods have cloned fragments?

The first step in analyzing the effect of clones in changeability is to establish if there is any difference between the changeability of methods with clones and the changeability of methods without clones. Before analyzing the distributions of the changeability metrics, we make a normality test to establish which statistical test is appropriate to validate the dissimilarity of changeability in methods with and without clones. We found that none of the distributions of changeability measurements is normal. Therefore, as explained in the Appendix B, the appropriate test is the Wilcoxon test for the paired analysis i.e. the analysis of methods Sometimes Cloned when cloned, vs., when not cloned. Similarly, the Mann-Whitney test should be used for the non-paired analysis i.e. the analysis of methods Always Cloned vs. methods Never Cloned. The result of both tests is the p-value, a p-value below 0.05 would indicate that the distributions are likely to be different (the interpretation of the p-value is summarized in

Section 10.1.2.1). Table 10-5 and Table 10-6 summarize the p-values found for each metric, and each application.

Table 10-5. P-values for the similarity test between changeability metrics of methods AC vs. methods NC

	Freecol	JEdit	GanttProject	JBoss	Columba
Likelihood	0.0001789	0.00616	4.681e-11	0.004995	6.395e-14
Frequency	0.0002048	0.02579	2.638e-09	0.01704	1.945e-12
Impact	0.6164	0.06429	9.196e-05	0.9923	0.003154
Depth	0.0001167	0.03945	0.1548	4.008e-05	6.149e-06

The results shown in Table 10-5 indicate that the likelihood, frequency, and depth of AC-methods are different from those of NC-methods. However, there is no certainty on the difference in impact, given that for three of the five applications analyzed, the impact of change in AC-methods is similar to the impact of change in NC-methods. This is consistent with the results presented on Table 10-3 that show that the median of the impact is very similar between AC-methods and NC-methods.

Table 10-6. P-values for the similarity test between changeability metrics of methods SC, when cloned vs. when not cloned

	Freecol	JEdit	GanttProject	JBoss	Columba
Likelihood	7.99e-15	2.2e-16	2.452e-11	2.2e-16	2.2e-16
Frequency	8.627e-15	2.2e-16	8.302e-11	2.2e-16	2.2e-16
Impact	7.996e-15	2.2e-16	2.451e-11	2.2e-16	2.2e-16
Depth	7.998e-15	2.2e-16	2.452e-11	2.2e-16	2.2e-16

Results in Table 10-6, show that the changeability metrics when a method is cloned are, without a doubt, different from the changeability metrics when the same method is not cloned. Therefore, it is safe to say that having a cloned fragment inside a method affects the changeability metrics of that method.

10.2.2.2 Does changeability increase when methods have cloned fragments?

Once established that having a cloned fragment indeed affects the changeability of a method, it is important to establish if the changeability decay is higher or lower when the method is cloned. Given that, clones are supposed to have a negative effect on the maintenance of methods, we expect the changeability metrics to be higher when cloned.

→ *Comparison of quartiles*

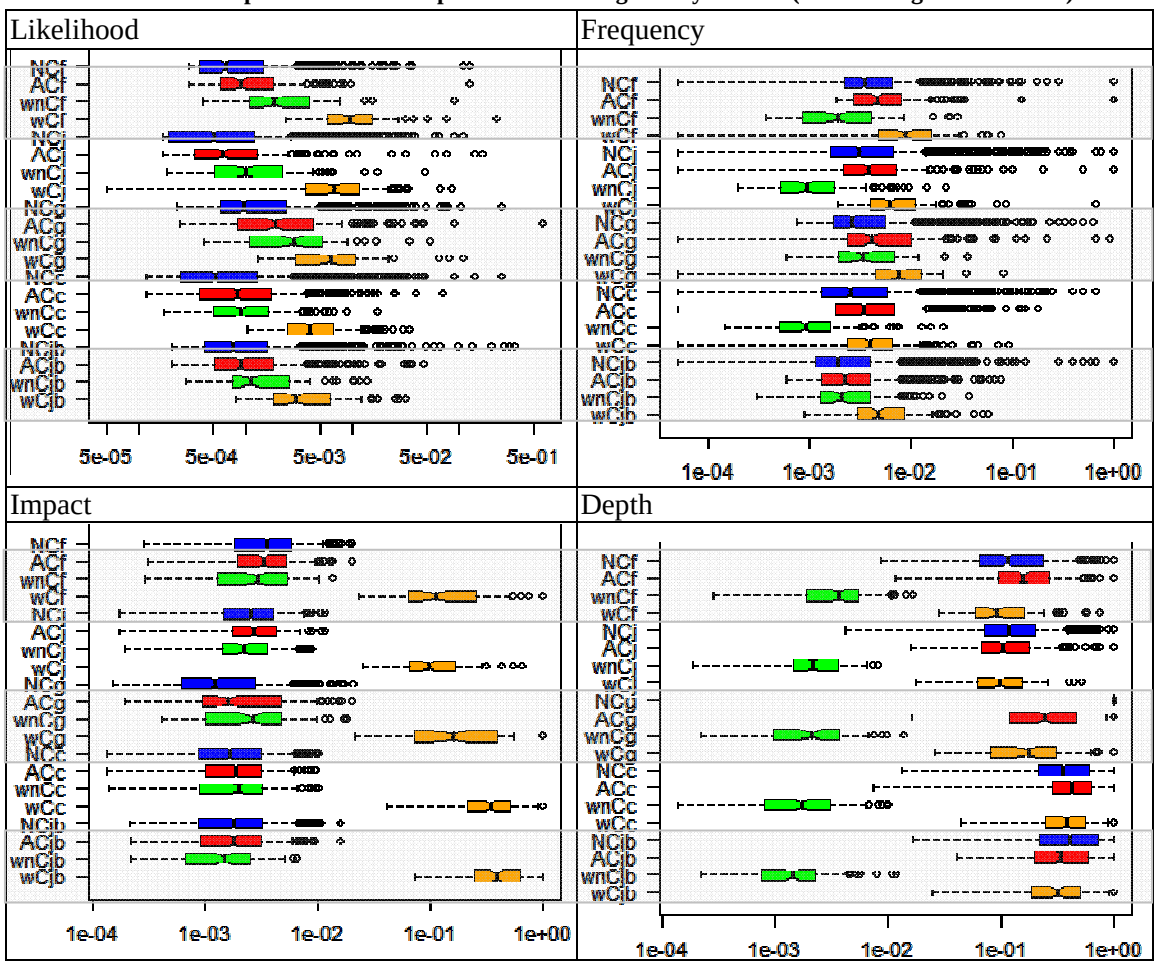
Here we depict key values of each distribution to give an idea of which values are higher, the ones when methods have cloned fragments, or the ones when methods do not have cloned fragments. The key values are represented using box plots, the meaning of these values are explained in Appendix B.

Table 10-7 shows the box-plots that compare each changeability metric for each method type in the following order (from top to bottom): NC-methods (in blue), SC-methods when not cloned (in green), SC-methods when cloned (in orange), and for AC-methods (in red). The order of the sets of methods was chosen by exposure to clones. In that way, if clones are correlated with changeability decay, the value of the metrics would be higher than the previous set of methods. From the top of the graph to the bottom of the graph it is compares the metrics for Freecol, JEdit, Ganttproject, Columba and JBoss. The box-plots of Freecol, Ganttproject, and JBoss are covered by a gray shadow as a visual aid to differentiate the box-plots of different applications. Note that the metrics of AC-methods and of NC-method have many more outliers than the metrics of SC-methods, especially NC-methods.

Based on the graphs on Table 10-7 we can conclude that methods that have had clones have a higher likelihood than NC-methods, those with the highest likelihood are SC-methods when not cloned, followed by SC-methods when cloned, and by AC-methods. This indicates that cloning indeed affects the likelihood of changes as suggested previously (see Figure 9-13, Table 9-2). In addition, the frequency of change for AC-methods is slightly higher than the frequency of changes for NC-methods. The methods with the highest frequency were those in which the method had a cloned fragment (i.e. SC-methods when cloned), and the methods with lowest frequency of changes were SC-methods when not cloned. There is no much difference in the impact of cloned methods vs. without clones, unless the clone in the method was deleted. Finally, that AC-methods have a slightly higher depth than NC-methods, SC-methods have a depth in between NC and AC-methods, and SC methods when not cloned tend to have a very low depth.

However, one should keep in mind that these results do not cover any cloned methods, given that, in order to analyze changeability it is necessary that the method present changes and just the minority of cloned methods changed. In average just 20% of AC-methods changed, and less than 10% of SC-methods changed.

Table 10-7. Comparison of the box-plots for the changeability metrics (x-axis in logarithmic scale).



In conclusion, we cannot identify whether or not AC-methods have higher impact, or depth than NC-methods based only on the analysis of box-plots. However, they tend to have slightly higher frequency and likelihood. This is not the case when comparing SC-methods. All the metrics seem to be higher when the method does not have a cloned fragment, which is contrary to expectations but consistent with the median description of the metrics presented in section 10.1.1. In general, it seems that the worse changeability values for likelihood, frequency, and impact are in SC-methods when not cloned. Regarding the depth, SC-methods tend to change with closer methods than AC-methods or NC-methods, in particular when they are cloned.

→ *Comparison of distribution graphs*

Distribution graphs depict on the x-axis all the possible values of an item set, ordered from the

lowest to the highest, and in the y-axis it depicts the amount of items that has that value from the item set. Therefore, the peaks in the curve indicate the most popular values, and the width of the curve indicates the spread of the values in the curve. When two distributions are compared, they are plotted in the same graph. The distribution that has higher values would have its peak to the right of the other distribution.

Table 10-8 shows the comparison of the distributions When Cloned, depicted with gray shapes; and When Not Cloned, depicted with black contours. The x-axis has the values of the metrics, and the y-axis has the percentage of methods that had that value for the period with clones, or without clones. The graphs make evident that for all the metrics, and for all the applications, more methods had their When Not Cloned metrics closer to zero than their When Cloned metrics. In fact, the spread of the distribution of metrics When Cloned is much higher than the width of the distribution of metrics When Not Cloned, which tends to have very small values. These graphs support the hypothesis on the harmfulness of clones, as well as previous results, i.e. the comparison of distributions by median value (section 10.2.1), and by box-plots (section 0). Therefore, metrics When Not Cloned are higher than metrics When Cloned for SC-methods.

Table 10-8. Comparison of the distributions of changeability metrics for methods SC: when cloned (gray) vs. when not cloned (black).

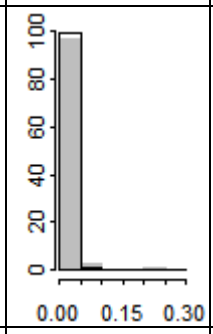
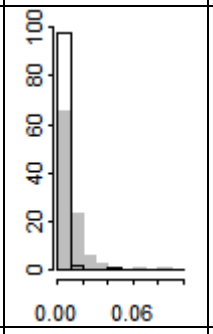
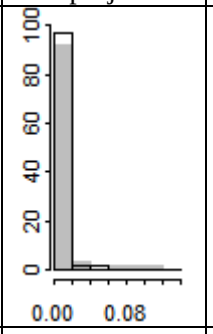
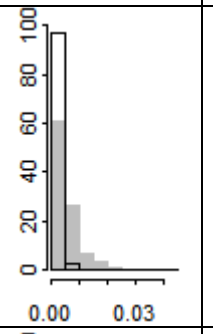
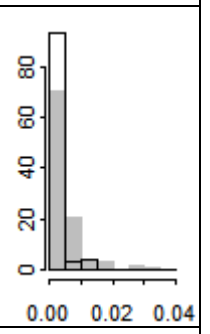
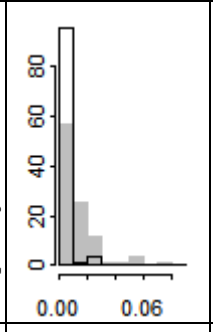
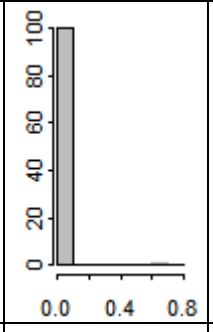
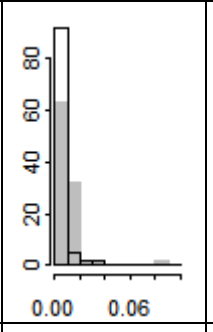
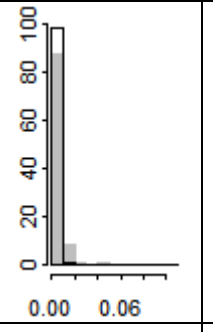
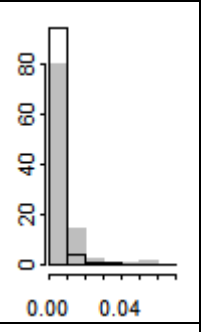
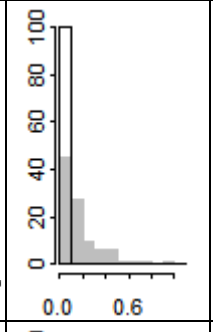
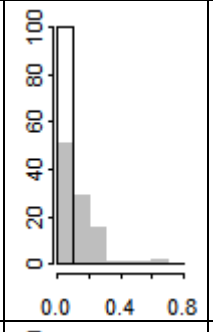
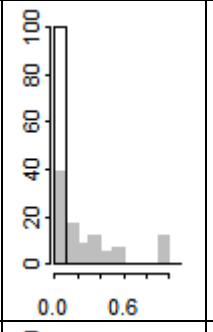
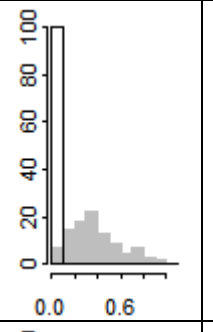
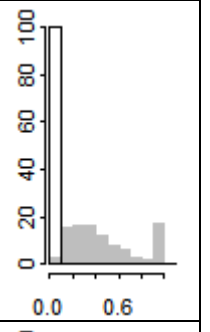
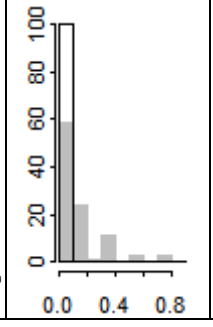
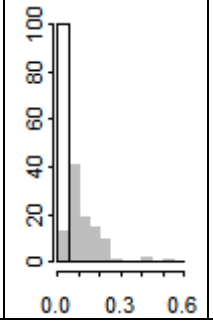
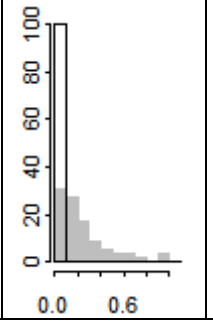
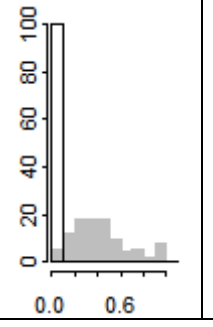
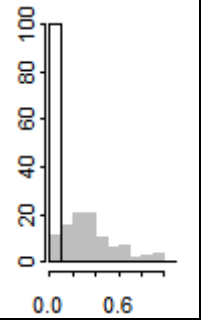
	Freecol	JEdit	Ganttproject	Columba	JBoss
Likelihood					
Frequency					
Impact					
Depth					

Table 10-9. Comparison of the distributions of changeability metrics for methods AC(gray) vs. NC(black)

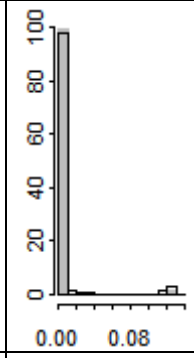
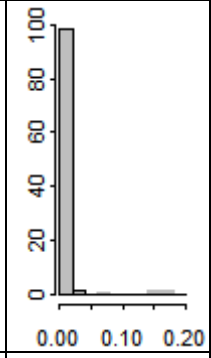
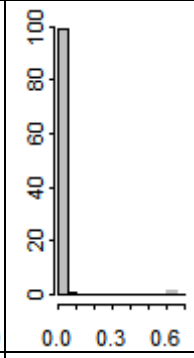
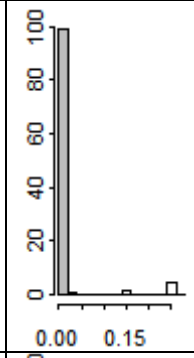
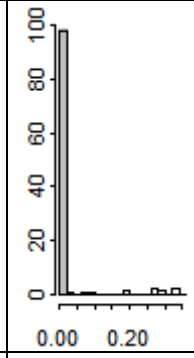
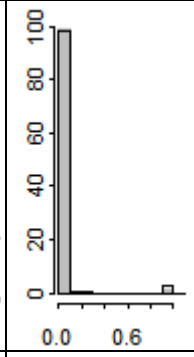
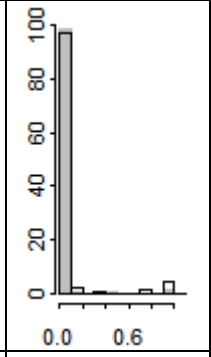
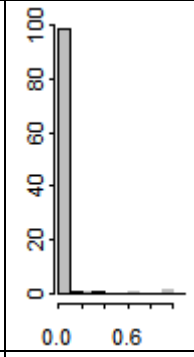
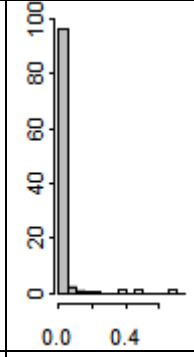
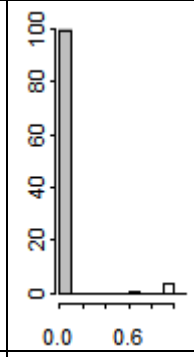
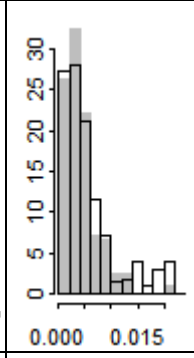
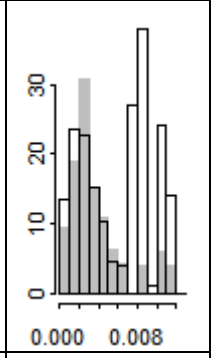
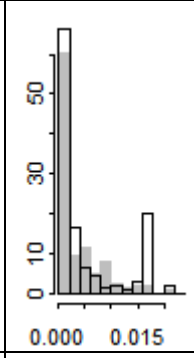
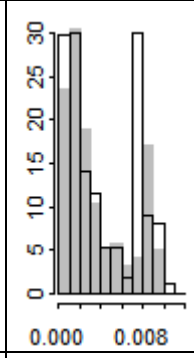
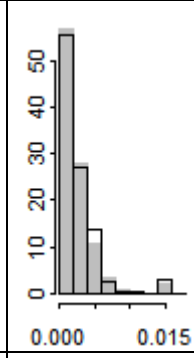
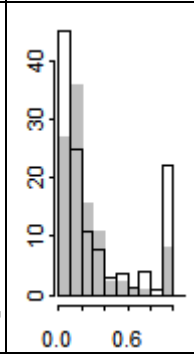
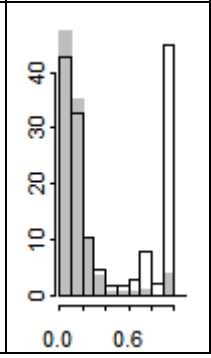
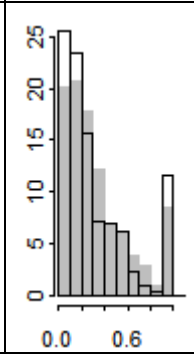
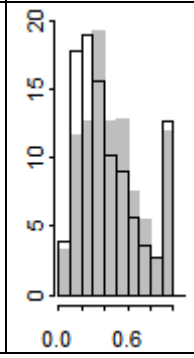
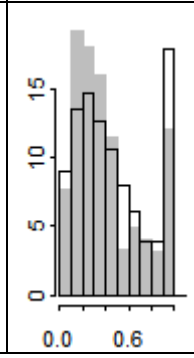
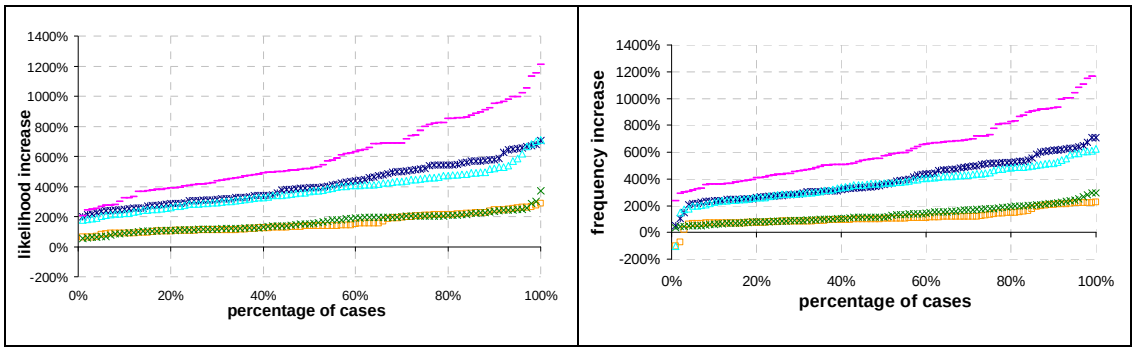
	Freecol	JEdit	Ganttproject	Columba	JBoss
Likelihood					
Frequency					
Impact					
Depth					

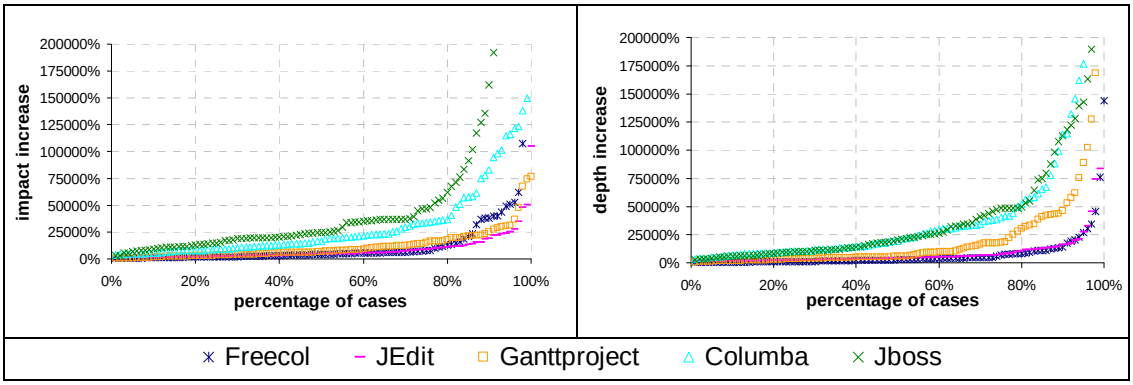
Table 10-9 show that the metrics Always Cloned are very similar to the metrics Never Cloned, which is consistent with the comparison of distributions by median value (in section 10.1.1), and by box-plots (in section 0). Although the static tests shown that the likelihood, frequency, and depth are different, in the graphical representation of likelihood and frequency show very few differences.

→ *Graphs of increase of changeability metrics when SC-methods have clones*

Whenever there are doubts of which distribution presents higher values, the last option that can be used is to calculate the difference of the metrics when cloned with the metrics when not cloned. The evidence so far supports the hypothesis that the metrics when cloned are larger. The increase of changeability metrics in the cloned periods is presented in Table 10-10 (see section 0). The *y*-axis shows the level of increase of the metric and the *x* axis shows the percentage of methods that presented at most that metric increase. If the metrics when cloned are higher than the metrics when not cloned, the distribution graph of the difference of the metrics will have most of its values above zero. This means that most of the metrics when cloned have a greater value than the metrics when not cloned. Conversely, if the metrics when cloned are lower than the metrics when not cloned, the distribution graph of the difference of the metrics will have most of its values below zero.

Table 10-10. Increase of metrics when cloned. X-axis: percentage of SC-methods, Y-axis: increase





The figures in Table 10-10 show that all SC-methods presented higher metrics when cloned, that is, their changeability decayed when cloned. In fact, the figures seem to confirm the intuition that clones *can be* very harmful for changeability. The metrics that assess the amount of changes (i.e. likelihood and frequency) tended to increase up-to one order of magnitude. The metrics that assess the complexity of changes (i.e. impact and depth) tended to increase up-to three orders of magnitude.

10.2.3 Identification of characteristics that affect changeability

Having established that cloning is correlated with changeability, that AC-methods have a slightly higher changeability decay than NC-methods, and that SC-methods behave inversely to AC-methods and NC-methods presenting higher changeability metrics when cloned than when not cloned; it is possible to investigate which factors are correlated with high or low changeability. This section presents method characteristics, and clone characteristics that are correlated with changeability and whether the relation is inverse or direct. All the characteristics analyzed are expected to have a direct relation with the changeability of the method.

10.2.3.1 Characteristics to analyze

The characteristics analyzed for all methods are:

Commit created: Commit transaction in which the method was created

NOP: Number of parameters

LOC: Lines of code of the method

Complexity: Number of branches of the method

Fan-in: Number of methods that call the method

Fan-out: Number of methods called by the method

Was cloned: Say if the method had cloned fragments at any point in its lifetime

The characteristics analyzed for cloned methods are:

Clone size: Number of tokens cloned

Family size: Number of fragments that compose the family

No. of families: Number of clone families with which the method is related (by its cloned fragments)

Commit created: Commit transaction in which the method was created

Lifetime affected: Percentage of commit transactions in which the method was cloned

Percentage affected: Percentage of tokens cloned from the tokens of the method

%Syntax tokens: Percentage of syntax tokens in the clone that identifies the family

%Literals: Percentage of tokens that refer to literals in the clone that identifies the family

%Dissimilarity: Percentage of tokens that are different between the cloned fragment and the clone that identifies the family

%Method-Type different: Percentage of differences in tokens referring to method calls and types between the cloned fragment and the clone that identifies the family

%CCM: Stands for Changes due to the Clones in a Method. It is the rate of the number of changes inside the cloned fragments of a method, over the number of changes of that method.

%CCF: Stands for Changes due to the Clones in a Family. It is the rate of the number of changes inside the cloned fragment of a method, over the number of changes of the family corresponding to that method. In case a method has fragments of several clone-families, the %CCF is calculated as the average CCF for all its fragments.

%CDF: Stands for Changes Divergent in a Family. It is the rate of the number of **divergent changes** inside the clone-families related to the method, over the number of changes of the family corresponding to that method. A **divergent change** is a change in a clone-family that affects only some of its cloned fragments. In case a method has fragments of several clone-

families, the %CDF is calculated as the average CDF for all its fragments.

%CDM: Stands for Changes Divergent in a Method. It is the rate of the number of **divergent changes** inside the clone-families related to the method, over the number of changes of that method. A **divergent change** is a change in a clone-family that affects only some of its cloned fragments. In case a method has fragments of several clone-families, the %CDM is calculated as the average CDM for all its fragments.

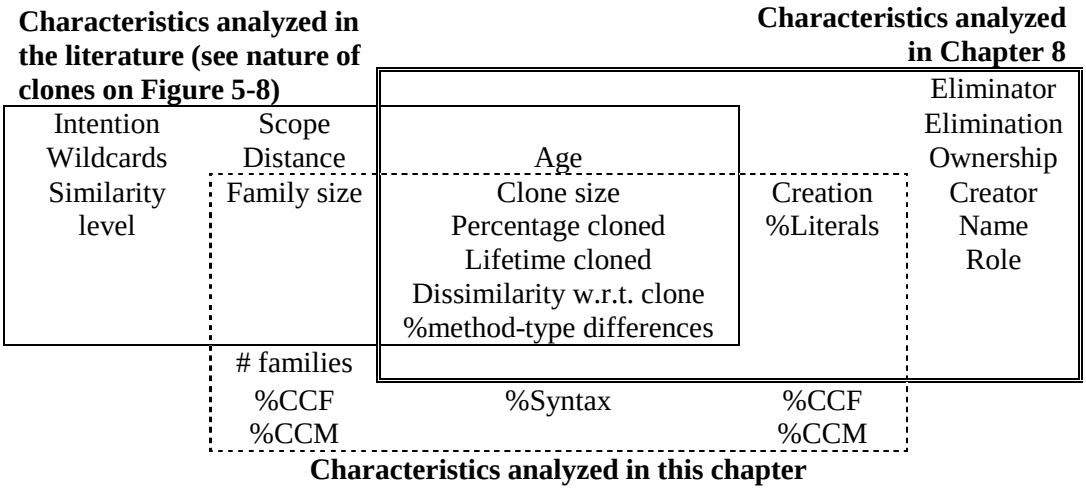


Figure 10-14. Characteristics analyzed in the literature vs. characteristics analyzed in this chapter

Figure 10-14 presents the set of characteristics analyzed in this document. The intersection between the characteristics analyzed in the literature, and those analyzed in Chapter 8 are commented in section 8.2. The characteristics analyzed in this chapter are considered for two reasons: first, they may be strongly related with the effect of a clone on the changeability of the method that hosts it, and second, they have numerical values that can be calculated automatically. Note that the eliminator and elimination values would not give us much information about the effect of the clone because in most of the cases the methods remain cloned until they are deleted. The ownership also gave very few results to be a good discriminator of the harmfulness of clones. The results on the creator indicate that there might not be different cloning behaviors per developers. Finally, the role and the name are not numerical values.

In this chapter, we added six characteristics to those presented in chapter 8. The size of the family was not considered as a characteristic to analyze in Chapter 8 because it was easy to

notice in the graph visualization, which showed that most of the clone families only had two members. However, if a method has a clone that belongs to a large family it is likely that it would require complex changes in order to keep the whole family consistent. Note that the size of the family is different to the number of families. If a method has several clones, all of them belonging to different clone families, it would require consistent changes whenever any of the clones of any of the families changes.

The percentage of literals has a similar motivation that the percentage of method-type differences in the clone fragment with respect to its family. If a clone has too many literals, it might be an accidental clone. Given that accidental clones are fragments of code similar by chance, they would not require consistent changes, therefore they would not increase the changeability decay of the methods that host them. Similarly, a high percentage of syntax tokens would indicate structural clones. Structural clones are also accidental clones because they indicate structures commonly used in a programming language. For instance, Java has an idiom for traversing a collection. A very common clone is traversing a collection and updating its values or calling a method for each one of the elements in the collection.

The creation commit of the first clone of a method would indicate the accumulation of the effects of having clones over time.

Finally, given that clones are believed to be harmful because of the need of convergent changes, we decided to calculate the ratio of changes due to a clone per method (%CCM), and per family (%CCF), as well as the ratio of divergent changes per method (%CDM), and per family (%CDF). If these ratios are high, we expect that the changeability decay of the method that hosts the clone increases. These characteristics are not in Chapter 8 because its aim is describing clones found, without regarding their changes which are tackled later.

10.2.3.2 Results of applying relative risk to correlate the characteristics to changeability

There are three sets of analyzed data to assess the correlation between clone and method characteristics and changeability (see Table 10-11). The first set is identified as *All*, this set contains all the methods that have ever changed. The changeability measurements for this set are calculated over the entire lifetime of the method. The second set is called *C*, this set contains all the methods that had a cloned fragment at any point in their lifetimes. All measurements (the changeability metrics, the method characteristics, and the clone characteristics) for the *C* set are calculated only for the period of time in which the method was cloned. The third set, called *SC*,

contains the methods that had a cloned fragment for a fraction of their lifetime. The characteristics analyzed comprise the period in which the method is cloned, but the metrics are calculated as the difference of the metrics when cloned minus the metrics when not cloned. In this way, a correlation would indicate that the characteristic exacerbates the effect of the clone in changeability.

Table 10-11. Sets of methods analyzed for correlations between characteristics and changeability

Set of methods analyzed	Description	Period of time taken into account
All	AC-methods + NC-methods + SC-methods	Characteristics & Metrics: all their lifetime
C (cloned)	AC-methods + SC-methods	Characteristics & Metrics: when cloned
SC (sometimes cloned)	SC-methods	*Characteristics*: for their period cloned *Metrics*: metrics for their period cloned minus metrics for their period not cloned

Given that, the results analyze the relative risk that a characteristic increases the chance of having higher changeability. It is necessary to define common thresholds to consider the value of a characteristic as high or low, and to consider the value of a changeability metric as harmful or not. The method used to calculate the thresholds consists of finding the median value of the 25^{th} and 75^{th} percentiles of the characteristics and metrics of each application analyzed. The thresholds obtained are presented in Table 10-12.

Table 10-12. Thresholds to evaluate the relative risk.

Changeability metric (All & C)	Harmful if above	
Likelihood	0.001686	
Frequency	0.006657	
Impact	0.003372	
Depth	0.4286	
Difference of changeability metric (SC)	Harmful if above	
Likelihood	0.0007568	
Frequency	0.003008	
Impact	0.001445	
Depth	0.1308	
Characteristic	Has a low value if below	Has a high value if above
Commit created	582	1494
Complexity	1	3
NOP	0	2
LOC	6	31
Fan-in	0	2
Fan-out	1	10
Was cloned	0	1
Clone size	39	67
Family size	2	6
No. of families	2	6
Commit cloned	366	1350
Lifetime affected (*time cloned*)	0.8225	1
Percentage affected (*%tokens cloned*)	0.1919	0.603
%Syntax tokens (*tdiff*)	0.04569	0.2092
%Literal tokens (*ldiff*)	0.03376	0.1178
%Dissimilarity (*%tokens diff*)	0.03008	0.1026
%Methods & Types diff (*mdiff*)	0	0.01794
%CCM	1.667	5
%CCF	1.333	5
%CDF	0.04655	1
%CDM	0.3	1

The results of applying the Relative Risk modification proposed in section 10.1.3.2 to evaluate the strength of the relation between characteristics and changeability decay are summarized in Table 10-13 and Table 10-14. Each cell in these tables could have three types of values:

- a hyphen, that indicates that the characteristic was not found to be related with changeability decay

- a capital D followed by a number, that indicates that there is a direct relation between the

characteristic and changeability decay. The number indicates the strength of the relation, if it is above 1.5 it is considered that there is a mild relation; if it is above 2 it is considered that it is a strong relation.

- a capital I followed by a number, that indicates that there is an inverse relation between the characteristic and changeability decay. The number indicates the strength of the relation, as explained above.

Table 10-13. Relative risk between method characteristics and changeability decay

	Likelihood			**Frequency**			**Impact**			**Depth**		
	All	**C**	**SC**	**All**	**C**	**SC**	**All**	**C**	**SC**	**All**	**C**	**SC**
Complexity	-	-	-	-	D 1.6	D 1.6	-	-	-	I 1.2	-	-
NOP	D 1.3	-	-	D 1.3	D 2.0	-	-	-	-	-	-	-
LOC	D 2.6	-	D 1.4	D 2.1	D 1.8	D 1.5	-	-	I 1.9	I 1.3	-	I 1.5
Fan In	-	-	I 1.4	I 1.2	-	-	-	-	-	I 1.1	-	I 1.5
Fan Out	D 1.9	-	-	D 1.7	D 1.7	D 1.9	-	-	-	I 1.4	-	-
Com. created	I 4.8			I 7.9			-			-		
Was cloned	D 1.3			-			-			-		

Table 10-14. Relative risk between clone characteristics and changeability decay

	Likelihood		**Frequency**		**Impact**		**Depth**	
	C	**SC**	**C**	**SC**	**C**	**SC**	**C**	**SC**
Clone size	-	-	-	-	-	-	D 1.2	-
Family size	-	-	-	-	-	-	-	-
No. of families	D 1.7	-	D 1.6	D 1.7	-	I 1.3	-	-
Commit cloned	-	-	I 4.6	I 4.2	-	-	D 1.1	-
Lifetime affected (*time cloned*)	-	-	D 1.0	D 1.0	-	-	-	-
Percentage affected (*%tokens cloned*)	-	I 1.4	I 1.3	I 1.5	-	D 1.9	-	D 1.7
%Syntax tokens (*tdiff*)	-	-	-	-	-	-	-	-
%Literal tokens (*ldiff*)	-	-	-	-	-	-	-	-
%Dissimilarity (*%tokens diff*)	-	-	-	D 1.5	-	-	-	-
%Methods & Types diff (*mdiff*)	-	D 1.3	-	-	-	I 1.3	-	-
%CCM	-	-	-	-	-	-	-	-
%CCF	-	D 2.1	-	-	-	-	-	-
%CDF	-	-	-	-	-	-	-	-
%CDM	-	-	-	-	-	-	-	-

Strong relations in Table 10-13 and Table 10-14 are highlighted with a dark gray shadow, while

mild relations are highlighted with a light gray shadow.

According to Table 10-13, there are two types of relations between the characteristics and the metrics of changeability decay. The first type of relation is when the characteristic has the same relation for all metrics; for instance, fan-in and commit created are always inverse to the changeability metrics to which they are related. The second type of relation is when the characteristic has a different type of relation for different metrics. Usually one type of relation (direct or inverse) for the metrics related with the number of changes (i.e. likelihood and frequency); and the opposite relation for the metrics related with the complexity of changes (i.e. impact and depth). For instance, complexity, loc, and fan-out are directly related with likelihood and/or frequency; but inversely related with impact and/or depth.

The characteristics that are differently related with the number and complexity of changes indicate that methods that are complex / large change more, and in isolation or with very few close methods. They may also indicate that the decrease in the complexity of changes may counter-effect the increase in the number of changes. However, we think that the previous hypothesis is unlikely because the relations with the number of changes are stronger than the relations with the complexity of changes.

The following characteristics are related with changeability decay metrics. The characteristics are mentioned in order of strength of their relation.

The method-characteristics related with likelihood are: *commit created*, *loc*, *fan-out*, *fan-in*, and *nop*/*was cloned*. The clone-characteristics related with likelihood are *%CCF*, *number of families*, *percentage affected*, and *percentage of method/type tokens that differ from the family*.

The method-characteristics related with frequency are: *commit created*, *loc*, *fan-out*, *complexity*, *nop*, and *fan-in*. Frequency is related with the following clone-characteristics: *commit cloned*, *number of families*, *percentage affected*, *% of dissimilarity*, and *lifetime affected*.

Impact is inversely related to loc. The clone-characteristics related with impact are: *percentage affected*, and *number of families* / *percentage of method/type tokens that differ from the family*.

Depth is inversely related with *loc*, *fan-in*, *fan-out*, and *complexity*. In terms of clone-characteristics, depth is directly related with the *percentage affected*, *the size of the clone*, and the *commit cloned*.

Note that having had clones (i.e. was cloned) is the relation that has the weakest relation with

the changeability metrics. This may be explained by the fact that the changeability metrics for AC-methods is very similar to the metrics of NC-methods, so the only harmful clones would be SC-methods. In that sense, the characteristics related with the changeability difference for SC-methods could help to identify harmful clones, i.e. commit cloned, no. of families, percentage affected, etc.

10.2.3.3 Characteristics strongly related with changeability decay

Given that there are several characteristics analyzed but just a few of them presented strong relations with the changeability metrics, this section only presents those characteristics with a strong relation with changeability. The rest of characteristics are presented in the Appendix C. For each characteristic, there is also the results of the graphic correlation and a rationale for its relation with changeability, the explanation of the presentation of results is shown in of Figure 10-15.

The only characteristic that is presented, in spite that it did not show strong relations with any of the changeability metrics is *was cloned*, because it shows that being cloned alone is not a key factor in changeability decay. Note, in Table 10-13, that all the other characteristics of a method have a stronger relation with changeability decay than having had clones.

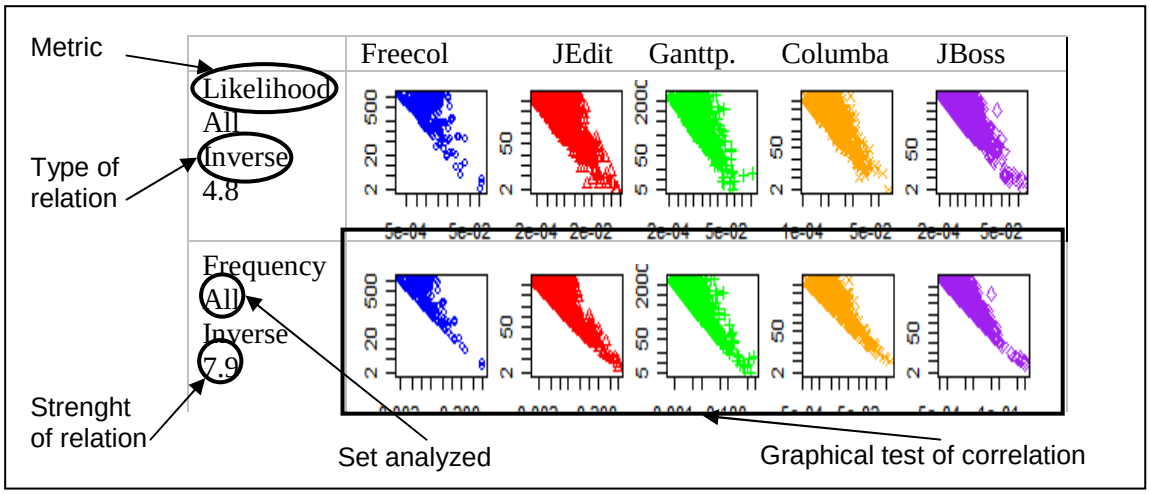


Figure 10-15. Explanation of results table

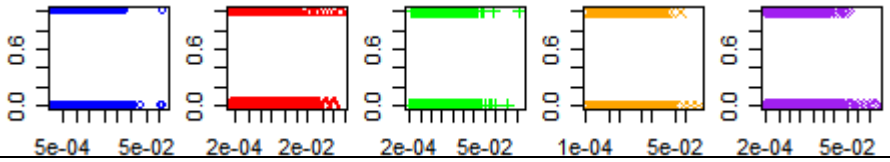
Each characteristic is presented as a table with its relation with changeability (see Figure 10-15). The results are presented as a table per characteristic analyzed. Each table has a row per changeability metric, with a set of data in which the characteristic was correlated to changeability. Each row has the average of the relative risks of all the applications analyzed, as

well as the graph that plots the value of the characteristic vs. the value of the metric for each one of the applications analyzed in the following order Freecol JEdit, Ganttproject, Columba, and JBoss. When any of the applications could not decide which relation existed between the characteristic and the metric, there is no row. If the characteristic and the changeability metric have a strong relation, i.e. the relative risk value is above two, the relation is highlighted in bold. The rest of the section presents the strong relations found between (method/clone) characteristics and changeability in the format of Figure 10-15.

→ *Changeability metrics vs. having had clones*

Having had clones increases the likelihood of changes (Table 10-15). However, note the relation is weak (with a relative risk of 1.3). This is consistent with the percentage of methods changed per group of methods, which showed that methods cloned are more likely to change (see Table 9-2). These results are interesting because they indicate that, regardless of the type of method, or of the type of clone, having had clones increases the chances of changing of a method. Furthermore, it is also interesting that no other changeability metric is correlated with the fact of being cloned. This could be due to AC-methods having a slightly higher, but very similar changeability than NC-methods; while SC-methods were the only ones that presented a significant increase in changeability decay. Given that SC-methods are the minority of the methods analyzed (approx. 8%), their effect in other changeability metrics might not be enough to correlate them with these tests.

Table 10-15. Relation between changeability metrics and the fact that the method was cloned or not

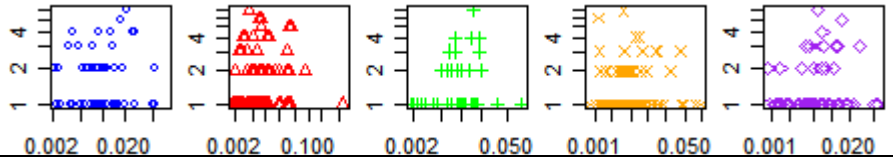
	Freecol	JEdit	Ganttproject	Columba	JBoss
Likelihood All Direct 1.3					

→ *Changeability metrics vs. number of parameters of the method*

The number of parameters is directly related with the frequency of changes in SC-methods (see Table 10-16). This means that the higher the number of parameters, the more frequently the methods would change. A possible explanation for this relation is that methods that have a high number of parameters are avoiding the creation of an object, as Fowler explains for the bad smell called long parameter list [Fowler '99]. These methods are hard to understand, and

difficult to use; therefore they are more likely to produce inconsistent changes that require fixes later.

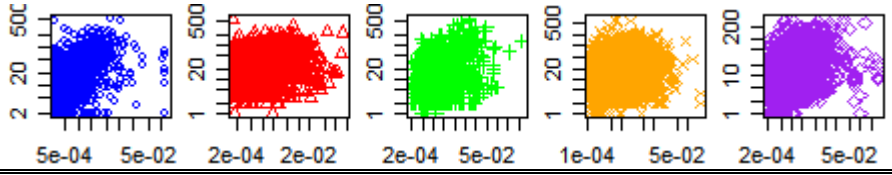
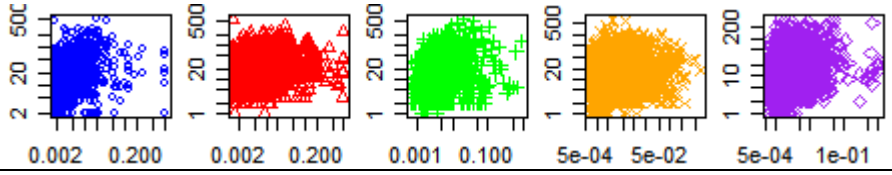
Table 10-16. Relation between changeability metrics and the Number Of Parameters the method

	Freecol	JEdit	Ganttproject	Columba	JBoss
Frequency Cloned Direct 2.0					

→ *Changeability metrics vs. size of the method*

Large methods change more, and more frequently (Table 10-17). A possible explanation for this finding is that large methods are more likely to implement several or more complex functionalities, and therefore are more likely to require changes frequently.

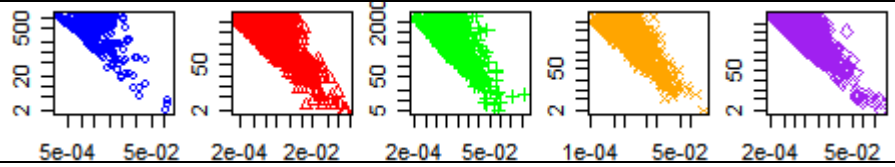
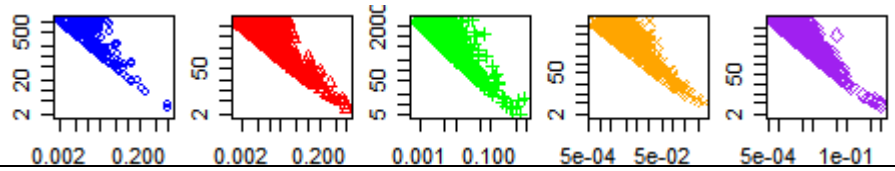
Table 10-17. Relation between changeability metrics and the Lines Of Code of the method

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood All Direct 2.6					
Frequency All Direct 2.1					

→ *Changeability metrics vs. commit in which the method is created*

According to the results (Table 10-18), methods created earlier in the application's lifetime tend to have a larger likelihood and frequency; inversely, those created late tend to have smaller likelihood and frequency. Therefore, the methods that compose the initial codebase of the application tend to change more, and more frequently. A possible explanation for this finding is that at the beginning of the application's history, there are uncertainties on how to distribute responsibilities across the application, which may cause more changes to the methods created earlier.

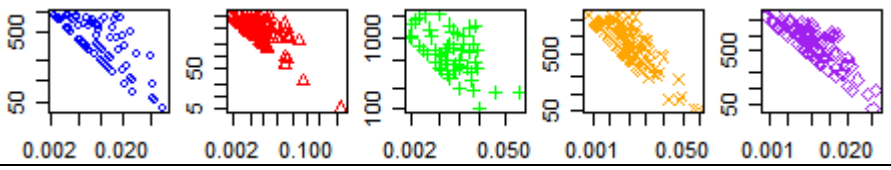
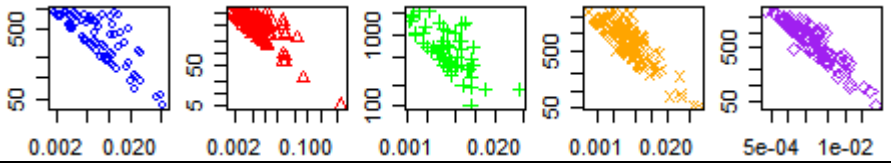
Table 10-18. Relation between changeability metrics and the commit in which the method is created

	Freecol	JEdit	Ganttproject	Columba	JBoss
Likelihood All Inverse 4.8					
Frequency All Inverse 7.9					

→ *Changeability metrics vs. commit in which the method becomes cloned*

Methods cloned early in the lifetime of the application are more frequently changed than methods cloned later in the lifetime of the application (Table 10-19). This could be explained because the earlier they are cloned, the more changes of hidden relations they have to check to verify if their version of the clone should be also changed. Furthermore, as shown with the commit in which the method is created, at the beginning of the application's history, changes are more frequent, probably because developers are not certain on how to implement the required functionality.

Table 10-19. Relation between changeability metrics and the commit in which the method becomes cloned

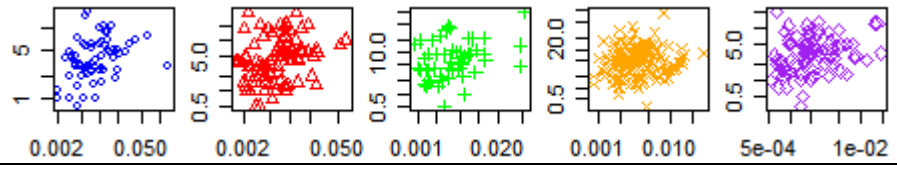
	Freecol	JEdit	Ganttproj.	Columba	JBoss
Frequency Cloned Inverse 4.6					
Frequency SC Inverse 4.2					

→ *Changeability metrics vs. percentage of changes with the clone family, of the changes in the clone family*

The amount of changes inside the fragments of a family is directly related with the chance of being changed (higher likelihood and frequency) as Table 10-20 shows. This coincides with the

belief that clones are harmful because they require consistent changes with their clone-families.

Table 10-20. Relation between changeability metrics and the percentage of changes with the clone family

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood SC Direct 2.1					

By presenting together the graphical results and the relative risk results, it is clear that the graphical relations alone are difficult to interpret. However, used in conjunction with the results from relative risk, they provide insight on the reasons for having weak relations, for having divergent results, and for applications in which it is not clear if the relation is direct or inverse. We have shown that the majority of the relations are weak, which should be taken into account to avoid predictions based on inaccurate correlations. For every strong correlation found, the chapter discusses possible reasons to support such finding, the rest of the relations analyzed are discussed in Appendix C.

This section has shown that the following characteristics have a very strong relation with the changeability decay of a method: lines of code, fan-out, and the percentage of changes inside the cloned fragment. Other characteristics that seem associated with changeability decay are: complexity, number of parameters, the size of the clone, the number of families related to the method, the size of the family, and the percentage of changes with the clone family.

The following characteristics have a very strong relation with changeability improvement: the commit in which the method is created, the percentage of tokens cloned, and the commit in which the first clone is introduced. A weaker relation with changeability improvement was also found for: the percentage of lifetime cloned, the percentage of differences in methods/types called in the clone, the percentage of syntax tokens in the clone, the percentage of tokens different in the clone, the percentage of changes in the cloned fragment but not in the family, and the percentage of changes in the family but not in the method.

Finally, the characteristics with strong relations are likely to be useful to predict when a cloned fragment inside a method will increase the changeability decay of the method.

10.2.4 Classification of methods by changeability measurements

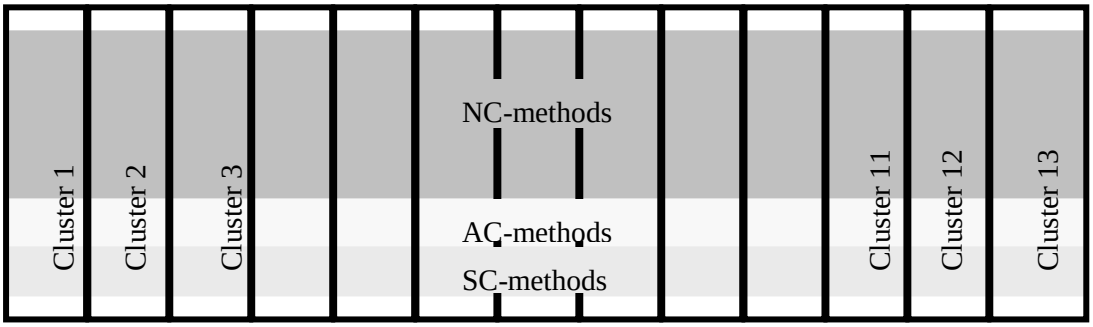


Figure 10-16. Partition of methods by changeability decay

This section aims to find levels of changeability in the methods analyzed (i.e. in those that changed from the sets AC, NC, and SC), to check if the methods with the same changeability level share characteristics or not. By changeability level we mean all conjunctions of changeability measurements that conduce to the same overall changeability. If methods with similar changeability have characteristics in common, the characteristics of the methods (and of the cloned fragments inside the methods) may help to identify whether or not a clone would increase the changeability decay of a method or not.

10.2.4.1 Method clustering according to their changeability

Clustering is a statistical analysis technique that assigns objects into groups. Clustering algorithms are based on the concept of similarity that establishes the distance among objects. The groups are formed from sets of objects that are close among themselves but far from other objects.

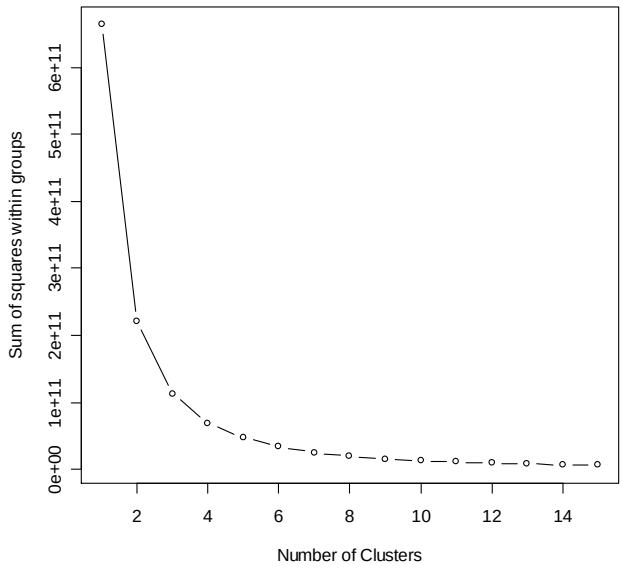


Figure 10-17. Average dissimilarity inside clusters per number of clusters.

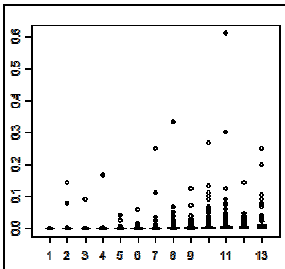
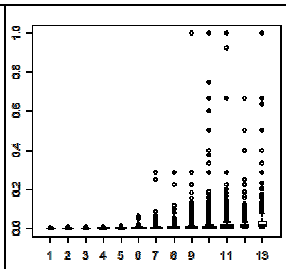
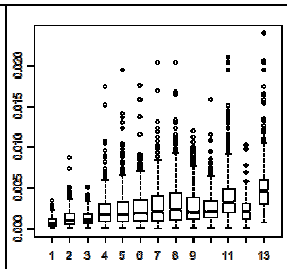
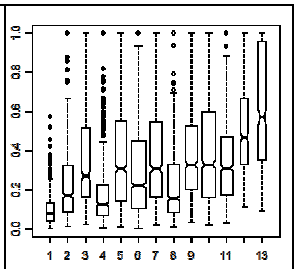
We used the clustering algorithm CLARA[21] implemented in the statistical package R[22], to group all the methods analyzed by their changeability metrics. CLARA is a partitioning clustering algorithm; this means that it requires the user to state the number of clusters to extract. In order to find the appropriate amount of clusters that are inside the set of methods, we plotted the sum of squares of the groups within the dataset against the number of clusters extracted. The sum of squares indicates how different the objects inside any of the clusters would be; therefore, the lower sum of squares, the better the number of clusters is. Figure 10-17 shows the sum of squares depending on the number of clusters when applied to the set of changeability metrics. Note that there is no reduction of the sum of squares from the 13th cluster on. Therefore, we decided to partition the data into 13 clusters.

We organized the clusters in such a way that the number that identifies the cluster also indicates the changeability level of the cluster. Note that each cluster has a higher changeability for each metric than its predecessor does, so, the overall changeability level of each cluster is higher than the changeability level of its predecessor (see figures on Table 10-21).

[21] Available at:: http://cran.cnr.berkeley.edu/web/packages/cluster/

[22] Available at:: http://www.r-project.org/

Table 10-21. Comparison of the box-plots of changeability metrics per cluster. The larger the id of the cluster, the higher the changeability decay level.

			
Likelihood per cluster	Frequency per cluster	Impact per cluster	Depth per cluster

10.2.4.2 Characterization of clusters according to methods or clones characteristics

We used the classification trees implemented in R to describe the method characteristics required to be part of a particular cluster. A classification tree is partition algorithm that aims to predict an outcome, in this case the cluster, based on the rest of the values obtained for an object. The clusters are the input for the classification trees, since the classification trees will aim to predict the cluster of a method based on its other characteristics. In this way, the classification trees provide the set of common characteristics of the methods that belong to a cluster.

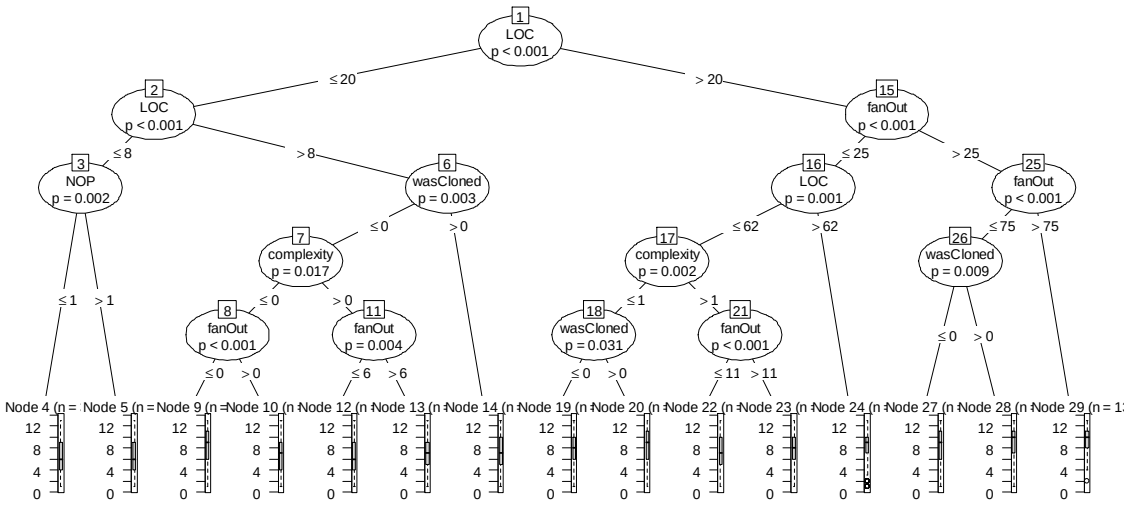


Figure 10-18. Classification tree of the clusters using characteristics of methods.

Figure 10-18 contains the prediction of clusters using the characteristics of the methods. The classification algorithm found fifteen nodes, whose characteristics are summarized in Table

10-22 (the nodes are organized by ascending clusters that they predict):

Table 10-22. Characterization of methods of different clusters, ordered by changeability decay level.

Clusters predicted	Nodes involved (characteristics of the node)
From 4 to 9, median at 6	Node 4 (*LOC* <= 8, *NOP* <= 1) Node 5 (*LOC* <= 8, *NOP* > 1) Node 12 (*LOC* > 8, *wasCloned* <= 0, *complexity* > 0, *fanOut* <= 6)
From 4 to 9, median at 7	Node 10 (8 < *LOC* <= 20, *wasCloned* <= 0)
From 5 to 9, median at 7	Node 13 (8 < *LOC* <= 20, *wasCloned* <= 0, *complexity* > 0, *fanOut* > 6)
From 5 to 10, median at 7	Node 14 (8 < *LOC* <= 20, *wasCloned* > 0) Node 22 (20 < *LOC* <= 62, *complexity* > 1, *fanOut* <= 11)
From 6 to 10, median at 8	Node 19 (20 < *LOC* <= 62, *fanOut* <= 25, *complexity* <= 1, wasCloned <= 0) Node 23 (20< *LOC* <= 62, 11 < *fanOut* <= 25, *complexity* > 1)
From 6 to 11, median at 9	Node 9 (8< *LOC* <= 20, *wasCloned* <= 0, *complexity* <= 0, fanOut <= 0) Node 20 (20 < *LOC* <= 62, *fanOut* <= 25, *complexity* <= 1, wasCloned > 0) Node 27 (*LOC* > 20, 25< *fanOut* <= 75, *wasCloned* <= 0)
From 7 to 10, median at 9	Node 24 (*LOC* > 62, *fanOut* <= 25)
From 7 to 11, median at 10	Node 28 (*LOC* > 20, 25 < *fanOut* <= 75, *wasCloned* > 0)
From 8 to 11, median at 10	Node 29 (*LOC* > 20, *fanOut* > 75)

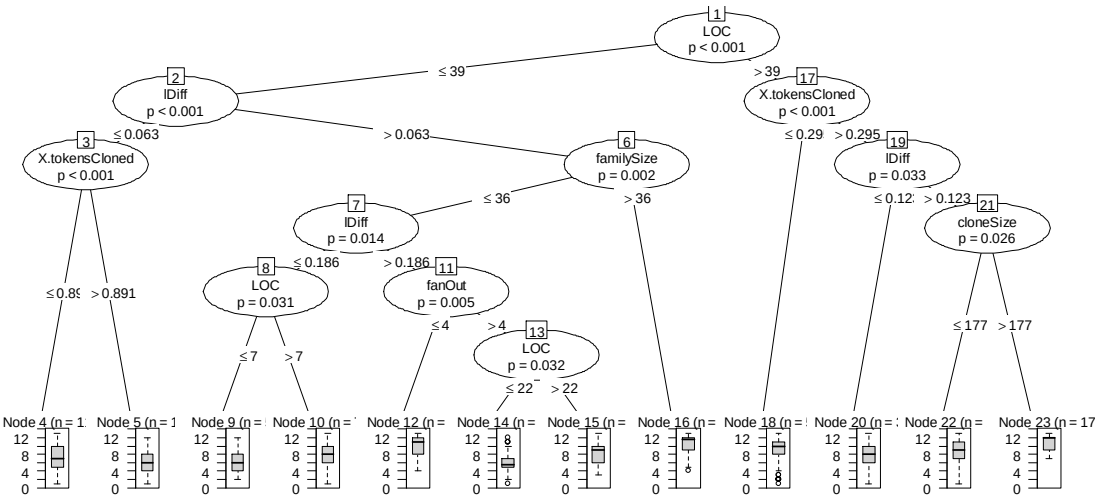


Figure 10-19. Classification tree of the clusters using characteristics of methods, and characteristics of clones.

Table 10-22 shows that it is not possible to predict the changeability of methods just by looking

at one of its characteristics. However, results suggest that a high number of lines of code (*LOC*), and a high fan-out (*fanOut*) are related with higher changeability decay.

Figure 10-19 contains the prediction of clusters using the characteristics of the methods, and the characteristics of the clones. This prediction was calculated with the methods that had a clone at least once in their lifetimes, and changed at least once. The classification algorithm found twelve nodes, whose characteristics are summarized in Table 10-23 (the nodes are organized by ascending clusters that they predict):

Table 10-23. Characterization of cloned methods of different clusters, ordered by changeability decay level.

Clusters predicted	Nodes involved (characteristics of the node)
From 4 to 8, median at 6	Node 5 (*LOC* <= 39; *lDiff* <= 0.06; *%tokensCloned* > 0.89) Node 9 (*LOC* <= 7; 0.06 < *lDiff* <= 0.18; *familySize* <= 36)
From 5 to 7, median at 6	Node 14 (*LOC* <= 22; 0.06 < *lDiff* <= 0.18; *familySize* <= 36; *fanOut* > 4)
From 5 to 10, median at 7	Node 4 (*LOC* <= 39; *lDiff* <= 0.06; *%tokensCloned* <= 0.89)
From 6 to 10, median at 8	Node 10 (7 < *LOC* <= 39; 0.06 < *lDiff* <= 0.18; *familySize* <= 36) Node 20 (*LOC* > 39; *%tokensCloned* > 0.29; *lDiff* <= 0.12)
From 6 to 10, median at 9	Node 15 (22< *LOC* <=39; 0.06 < *lDiff* <= 0.18; *familySize* <=36; *fanOut* > 4)
From 7 to 11, median at 9	Node 22 (*LOC* > 39; *%tokensCloned* > 0.29; *lDiff* > 0.12; cloneSize <=177)
From 8 to 11, median at 10	Node 18 (*LOC* > 39; *%tokensCloned* <= 0.29)
From 8 to 12, median at 11	Node 12 (*LOC* <= 39; 0.06 < *lDiff* <= 0.18; *familySize* <= 36; *fanOut* <= 4)
From 9 to 12, median at 12	Node 16 (*LOC* <= 39; *lDiff* > 0.06; *familySize* > 36) Node 23 (*LOC* > 39; *%tokensCloned* > 0.29; *lDiff* > 0.12; *cloneSize* > 177)

Results indicate that characteristics of the method, like lines of code in the method, are more important in determining the changeability of the method than characteristics of the clone.

Characteristics that until now seemed irrelevant in the analysis, like the percentage of literals in the cloned fragment (*ldiff*), do differentiate the correlation between clones and changeability decay.

Note that obvious characteristics to consider a clone harmful are indeed correlated with higher clusters, and therefore with higher changeability. These characteristics include the size of the cloned fragment (*cloneSize*), the number of cloned fragments that compose the family

(*familySize*), and the percentage of tokens cloned in a method (*tokensCloned*). However, some of these characteristics require a very high value in order to predict a method in a high changeability cluster. For instance, it is required that the clone to be larger than 177 tokens, or that the number of fragments in a family to be more than 36. Nevertheless, notice that cloned methods were located in clusters of a higher value than the average method.

Finally, note that the analysis of relative risk and graphical correlation (in section 10.2.3) predicted several of the key characteristics that could be used to predict changeability decay, i.e. the lines of code in the method (*LOC*), the number of methods called by the method (*fanOut*), and the percentage of tokens cloned in a method (*tokensCloned*).

10.3 Summary

This phase of the methodology helps to identify the differences in the changeability behavior of the SCE when they have the SCI. The fact that all the applications behaved in the same way for each type of methods indicates that the modification in changeability is indeed related with the fact of being cloned.

The majority of methods were NC-methods, and from NC-methods the majority changed. Therefore, most of the methods analyzed were not cloned. Nevertheless, given that the chance of changing in cloned methods is higher; the difference on the size of the sets analyzed was not as large as the difference on the size of the sets.

We have shown that the changeability of cloned methods is different to the changeability of methods without clones. The likelihood, frequency, and depth are different for AC-methods in comparison with NC-methods, and for SC-methods when cloned in comparison with SC-methods when not cloned. The impact of changes is very similar between AC-methods and NC-methods, and very different between SC-methods when cloned and when not cloned.

As expected, AC-methods and SC-methods when cloned presented higher changeability metrics than NC-methods and SC-methods when not cloned respectively. In addition to that, the relation between the changeability metrics in AC-methods and NC-methods is similar to the relation between SC-methods when cloned and SC-methods when not cloned, which was the expected behavior. That is, cloned methods have higher changeability decay than not cloned methods. AC-methods have changeability decay metrics slightly higher than those of NC methods, especially for the metrics that assess the amount of changes (i.e. likelihood and frequency). SC-

methods when cloned have changeability decay metrics much higher than those of SC-methods when not cloned. These differences are clearly shown in the comparison of the changeability metrics using box-plots Table 10-7.

We have found that, from the methods analyzed, cloned methods tend to change more. Given that, they also tend to have larger lifetimes, their likelihood and frequency values tend to be higher than those of NC-methods. We also found that SC-methods have a very low depth of changes when not cloned, probably because they are changed in isolation by late propagations. The greatest increase of changeability of a set with respect to the other sets is the changeability of SC-methods when cloned for the likelihood and the impact, indicating that they have the worst changeability decay. It is likely that the extra changeability of SC-methods when cloned is related to their clones because the same methods have much lower changeability metrics when not cloned. However, it is unclear why changeability metrics for AC-methods are not as high as changeability metrics for SC-methods when cloned.

We compared the characteristics of the SC-methods and AC-methods. We found that SC-methods tend to be smaller in clone size and family size, have lower percentage in syntax tokens, and a lower percentage of tokens cloned of the tokens of the method. In addition to that, SC-methods tend to have higher divergent changes, and higher percentage of method/type tokens that are different. These characteristics could cause the method to require more changes to keep implementing a similar functionality than the methods that are cloned with it. In addition, their low percentage of syntax tokens and their high percentage of different methods and types are consistent with semantic clones caused by problems in the design or by lack of appropriate abstractions of the language. However, further analysis is required to confirm this hypothesis.

Very few characteristics showed a strong relation with the changeability of methods, and indeed some of these characteristics were related with those that distinguish harmful clones. Although single characteristics cannot be considered alone to identify harmful clones, their combinations of characteristics above certain thresholds seem to be better discriminators. The characteristics that have the more weight in the changeability of a method are those related to its amount of responsibilities i.e. LOC, NOP, and fan-out. Given that, the characterization of cloned methods by changeability decay use these characteristics, we conclude that method characteristics weight more than any clone characteristic in the effect of a clone in the changeability of a method.

Chapter 11. Conclusions and future work

This thesis proposes a methodology to evaluate the effect of a source code characteristic that is considered harmful (i.e. a SCI) in the changeability of a SCE. This chapter presents the strengths and weaknesses of the methodology, as well as an evaluation of its application to the case of cloned code.

This chapter is organized as follows. The first section contains a summary and an evaluation of the methodology. The evaluation is done by enumerating advantages and limitations of the methodology. The second section evaluates the results of applying the methodology to clones, at the level of methods. This section lists the contributions of this thesis and explains which of those contributions are the most important. The third section analyzes all the threats to validity that the methodology may have. The fourth and last section presents areas in which this work can be extended, as well as areas that could not be part of this work due to time constraints. This section includes a detailed summary of issues found when applying the methodology, as well as some proposals to eliminate such issues.

11.1 Evaluation of the methodology

This section summarizes the benefits and drawbacks of applying the methodology.

Among the general benefits of the methodology is having a documented protocol. This means that it permits tracking the origin of research questions; hypotheses; identification, elimination and differentiation of variables, hypotheses, or cases, and possible explanations for the results. Besides, having a methodology facilitates replication and generalization of results. The replication is favored because a methodology provides a framework to aggregate the results, and provides a procedure that can be followed systematically across experiments. The generalization is facilitated because the methodology is described as a standard template to guide the instantiation of the methodology, to indicate if adaptations are required, and to describe how to implement such adaptations. Although the methodology seems sometimes obvious, following it helped to ensure systematic analyses.

11.1.1 Summary of the methodology

The methodology presented is divided in six phases. Three of the phases are of analysis, that means that they are tackle research questions about the Source Code Issue Under Study. The rest of the phases are context phases, which means that they are not intended to answer directly any research questions but to provide evidence to inform the research questions tackled in the other phases. Context phases include data gathering, comparison of the applications to analyze, and a detailed description of the SCIUS. Analysis phases include description of the nature, evolution, and impact on changeability of the SCIUS.

The phase on data gathering describes considerations to ensure to apply the methodology, as well as a proposal on how ensure them. The comparison of the applications to analyze permits being aware of the limitations and generalizability of the results; but also of aspects that could explain differences in the results across applications. The description of the SCIUS permits establishing an exact definition of the SCIUS, and distinguishing between hypotheses about the harmfulness of the SCIUS, and current evidence to support such hypotheses.

The nature of the SCIUS describes typical instances of the SCIUS by coloring all the instances found depending on the value of their characteristics. The purpose of this phase is to find out if characteristics that are related with harmful effects are a common case among the instances of the SCIUS. For instance, clones are supposed to be harmful because, among other reasons, they introduce excessive code that reduces the understandability. However, we found that clones tend to be small and to cover most of the methods; therefore, their impact on understandability might be low. This phase is exploratory, and aims to support the findings of later analysis phases.

The evolution of the SCIUS provides a general overview on the effect of the SCIUS. The evolution of the SCIUS assesses how generalized or localized is the SCIUS (i.e. extension), if the SCIUS is a long-term issue or not (i.e. persistence), and how grave it is (i.e. stability). Depending on the results of this phase, the results of the impact of the SCIUS in changeability could be magnified or reduced.

Finally, the effect of the SCIUS in changeability says whether SCEs with the SCIUS have different changeability decay than the SCEs without the SCIUS. Furthermore, it can provide evidence to distinguish if the changeability decay is worse when SCEs have the SCIUS, and how worse it is. Finally, the phase identifies the set of characteristics that are likely to point out

harmless and harmful clones.

11.1.2 Advantages

The suspiSCIUS methodology offers several additional advantages.

It is detailed. The methodology provides a background of key concepts. These key concepts include stating definitions of the concepts to analyze (see section 5.1), and comparing similarities and differences of the SCIUS and of the applications to analyze (see section 6.1). These concepts facilitate the construction of arguments because they permit establishing reasons for explaining the results, in particular, for explaining results that diverge from the hypothesis.

It is extensible. The methodology is designed to be used on several SCI at different levels of abstraction (at method level and above). It is not tied up to a particular tool but it discusses the priorities of the requirements of the tools that can be used (See sections 2.3.1, 0, and 7.1).

It provides triangulation. The methodology analyzes several sources of information that include historical/change relations, structural relations, semantic relations (by name similarity) using diverse analysis techniques like graphical correlations, statistical comparisons, statistical correlations, clustering, analysis of frequencies, etc. For instance, deciding if the SCEs with SCIUS have higher changeability decay (i.e. section 10.1.2.2) is analyzed with boxplots, histograms, and graphs that plot the increase of changeability metrics for the periods with the SCIUS. Another example is the characterization of harmful clones (i.e. section 10.1.3) that is analyzed first by correlation of individual characteristics with changeability decay, and second by detection of discriminators for the changeability of methods.

It minimizes observer bias. Given that most of the data is automatically analyzed, it is unlikely for the researcher to, inadvertently influence the results. Although there are choices made by the researcher, such as the applications analyzed of the thresholds to detect high or low values in metrics; the methodology provides guidelines for taking those choices. Furthermore, the impact of these choices is reduced by specific steps in the methodology. For instance, the choice of applications analyzed is tackled by documenting their similarities and differences so it is possible to define the limitations of the results. Another example, is the choice of thresholds that is handled by providing an algorithm to choose the thresholds (see page 232), such algorithm must be followed regardless of the application or metric analyzed.

It is replicable. The methodology is written in a didactic way so other researchers can use it.

It is didactic because it provides a detailed description, and rationale of all its phases; but also, because it provides an example of each one of the steps proposed (which is the application of the phase to the analysis of clones at the level of methods). Moreover, given that the adaptation of the methodology is publicly available (from the tools developed to perform data collection, and analysis to the data obtained), the results can be replicated.

It aims for causal relations. The methodology is an accumulative process. Given that, the analyses of different phases are orthogonal; their results point out diverse relevant characteristics of the analyzed applications, of the SCEs, and of the SCIUS. This rich set of small findings permits to evaluate alternative explanations for the observed facts. Finally, thanks to statistical tests, the methodology permits discarding results that could have been found by chance due to similarities across the analyzed applications.

11.1.3 Limitations

The suspiSCIUS methodology also has some disadvantages.

It is centralized/data-driven. All the analyses depend on an accurate and complete *data-gathering phase*. Therefore, any problem or missing data in the collection or attribution of the history would have significant impact in the results. For instance, the detection of SCEs, their lifetime, or their changes are fundamental steps on the data collection phase. Another example is the generation of changeability metrics per SCE per period with SCIUS, without SCIUS, and during their entire lifetime; any error in this step would damage all the analyses that depend on these metrics, i.e. those shown in Chapter 10.

It is application-dependent. The quality of results is highly dependent on the quality of the applications that perform the *data analysis*. Any error in the tools that collect or analyze the data will taint the results. However, given that several analyses are applied on the same dataset, it is likely that inconsistencies in the results would point out errors on the analysis tool. For instance, if the changeability analysis shows that the SCIUS increases significantly the changeability of SCEs but the stability of SCEs with SCIUS show that these methods do not change more; it would indicate that one of their implementations has errors.

It is heavy. SuspiSCIUS is by no means a lightweight methodology; gathering and analyzing the data are time-consuming tasks that must follow strict guidelines on the way in which they are preformed. Therefore applying the methodology to several applications requires a significant

investment. However, this investment is placed mostly on development time, given that several steps must be done automatically.

It presents a narrow concept of harmfulness. The harmfulness of the SCIUS depends only on its effects on changes. The methodology may conclude that a SCI is harmless although it may hamper the evolution of the application in another way; for instance, by introducing bugs or degrading its architecture. However, this is unlikely given that other indicators of harmfulness are related with changeability. For instance, bugs would increase the number of changes (i.e. likelihood and frequency); while the degradation of the architecture would increase the complexity of changes (i.e. impact and depth).

Its historical analysis is sensitive to check-in practices. The way in which the SCM repository is used can substantially affect assumptions on which the methodology is based. In particular, the assumption that a check-in represents a logical change; which means that a check-in is the implementation of a part of a modification request. Examples of check-in practices that violate this assumption are programmers check-in before finishing any logical change; and programmers that work on several modification requests in parallel.

There are steps of phases that cannot be applied. Depending on the type of SCI analyzed and the level of granularity chosen for the SCE, some parts of the methodology cannot be applied. For instance if the SCIUS is defined at the same level of granularity of the SCE e.g. god methods analyzed at the level of methods, it is impossible to analyze the extension of the SCIUS in the SCE, because it is always 100%.

11.2 Effect of clones in maintainability

Since the presentation of the first tools capable of finding cloned code, cloning has been an active topic. Although there is no consensus on the definition of a clone, the empirical analysis of the effects of cloning is one of the topics in the area whose importance has increased recently. However, deciding if cloning is harmful or not is still an open issue.

We have contributed to the area with the following original findings:

1. Cloned methods tend to change more than other methods. Just 22% of the methods of the applications changed, from these methods 11% were NC-methods, and 11% were

cloned methods. However, given that NC-methods are a larger set than the sum of AC-methods and SC-methods (see Table 9-2). Furthermore, the evolution of the instability for cloned methods (see Figure 9-13), also showed that the majority of changes in the applications analyzed occur in cloned methods. This is an interesting finding because another empirical study [Krinke '08] found that lines of code cloned change less than lines of code that are not cloned. We think that the findings are different because the clones he analyzes have at least 15 lines of code, while our clones have at least 3 lines of code and we measure changes at the level of method.

2. The distribution of clone creation follows that of method creation. This is a conclusion comes from the evolution of the extension of cloning in the application (see Figure 9-3), given that, it does not fluctuates but the size of the application does.

3. Most methods become cloned at the same time, rather than being copies of a previous, common seed. The analysis of the role of the method when it becomes cloned (see Figure 8-14) showed that the case in which a method receives a cloned fragment from another method is rare. Clone relations tend to appear as twins (i.e. when the cloned fragment is added to all methods cloned in the same commit), or as seeds (i.e. when the method did not change when it became cloned). Seeds are frequently twins, this means that, groups of methods are created cloned.

4. Once a method is cloned, it will tend to remain cloned most of its lifetime. The analysis on the nature of cloned methods (Figure 8-11) shows that they tend to remain cloned most of their lifetime. The evolution of the extension of cloning inside methods (Figure 9-5) also showed that methods are cloned most of their lifetime, with a slow decrease over time. This is an interesting finding because a previous empirical analysis concluded that clones are volatile [Kim '05]. The findings are different because they analyze the lifetime of cloned fragments in cloned families. In contrast, we analyze the percentage of the lifetime of the method in which cloned methods have at last one cloned fragment. This difference is explained by the transformation of cloned fragments into new families, which is discussed in section 8.2.7.

5. Most methods are cloned at the beginning of their lifetime. This conclusion comes from the finding that most of the methods are created cloned (see analysis of roles in section 8.2.10), and before its first 100 commits alive (see analysis of the age in which methods

become cloned in section 8.2.11).

6. Clones tend to cover most of the method that hosts them. The analysis of the nature of cloned methods showed that cloned fragments tend to cover the majority of clones of the method (see section 8.2.7), which was confirmed by the evolution of the extension of cloning inside methods (see section 9.2.1.1).

7. Most of the changes in cloned method occur inside the cloned fragment. The analysis of stability of clones inside methods (section 9.2.3.1) showed that most of the clones in cloned methods inside cloned fragments.

8. The changeability metrics are different when the method is cloned. The statistical analysis that compares distributions of changeability metrics (in section 10.2.2.1) showed that AC-methods have different changeability metrics than NC-methods for likelihood, frequency, and depth. In addition, SC-methods with clones have different changeability metrics than SC-methods when not cloned for all changeability metrics.

9. The worse methods in terms of changeability are those sometimes cloned, i.e. those that stop belonging to a family, probably due to an incomplete or divergent change. This conclusion comes from the fact that SC-methods when cloned presented changeability metrics much higher than those of SC-methods when not cloned (see section 10.1.2.2). Moreover, the impact, likelihood, and frequency of SC-methods when cloned are higher than the rest of the methods (see Table 10-7).

10. The effect of a clone in the changeability of the method that hosts it depends more on the type of method than on the type of clone. Contrary to expectations, the characteristics that are always present for discriminating cloned methods by changeability level are those that related to the method (see section 10.2.4.2).

11.3 Validity of results

The results of an empirical study depend on its reliability and validity. The reliability indicates to what extent the results are consistent every time the experiment is performed. Reliability is achieved by providing and following a protocol for performing the studies that is clear, complete, and that considers several methods to assess a phenomenon. This methodology aims to be such protocol. The validity indicates to what extent the observations represent accurately

the reality analyzed. Therefore, the way in which the methodology proposes to assess harmfulness, the data analyzed, the assumptions on the data analyzed, and the process followed may influence the strength of the conclusions.

Validity and reliability are conflicting characteristics. Usually, the reliability is increased by limiting the variables considered; this reduces the validity by narrowing the context that the experiment analyzes. Similarly, the validity is increased by considering contexts close to reality, which reduces reliability because several variables may affect the results.

The purpose of this section is to document factors that may have affected the validity of the results.

11.3.1 Internal or construct validity

Internal validity says to what extent the variables manipulated (independent variables) may have caused the effects observed in the variables that were not manipulated (dependent variables).

The methodology proposed independent variables that are adequate for the effects that each phase aims to assess. The dependent variable in the analysis of the nature of clones is each one of the characteristics of cloned methods, which dictate in which values fall the majority and the minority of cloned methods. The dependent variable in the analysis of the evolution of clones is the commit transaction, which change the values of the persistence, stability, and extension of the clones in the application. The periods cloned or not cloned are appropriate indicators of changes on changeability due to clones. The characteristics of methods / clones indicate the relation between the type of method / clone and the changeability behavior of the method. The changeability cluster is an appropriate indicator of the methods that have a similar changeability and therefore for the characteristics these methods have in common.

11.3.1.1 Issues related to the information gathered

These issues refer to the quality of the information collected.

1. Extraneous variables or confounding factors: It occurs when variables that may affect the dependent variable, different to the independent variables, are not controlled. In these cases, the outcome is disguised by the effect of such variables. There are three strategies to tackle these issues: restriction, matching, or stratification. Restriction occurs when the subjects analyzed do not have confounding factors. Matching occurs

when there is paired analysis i.e. when each subject is evaluated under different values of the independent variable. Stratification occurs when the results are analyzed in two sets: for subjects having the confounding factors, and for those that do not. If the experiment design uses any of these strategies, the results should not be affected by this type of validity issue.

We think that we have tackled confounding factors first by taking into account the similarities and differences in the applications analyzed. Second, by performing paired analysis to compare the changeability of methods i.e. AC vs. NC, SC when cloned vs. SC when not cloned. Third by stratifying the analysis, for instance in the relative risk analysis three different data sets are evaluated (1) for all methods: their characteristics and changeability during all their lifetime, (2) for cloned methods: their characteristics and changeability when cloned, and (3) for SC-methods: their characteristics during all their lifetime and difference of changeability when cloned and not cloned. Another example of stratification is the analysis of characteristics per cluster: all methods are clustered according to their changeability during their entire lifetime and classified according to their characteristics; and cloned methods are clustered according to their changeability when cloned and classified according to the characteristics of the method and of the clones.

Nevertheless, the frequency of change may be affected by the fact that method creation is not considered a change but clone creation is considered a change. Therefore, when comparing methods when cloned vs. when not cloned the amount of commits in the periods could differ by one commit only; which is likely given the very low amount of changes per method. However, this pitfall would affect only SC-methods that become cloned in a commit different to the one in which they are created. Nevertheless, note that most of the methods cloned were created cloned so this issue would affect only a minority of SC-methods.

→ *Issues related to the measurements used*

Also called construct validity, it refers to the accuracy with which the measurements reflect the concept to analyze.

1. Inappropriate measurements to represent the effect to analyze: It occurs when the measurements used to determine the effects of the independent variable are incorrect.

Changeability measurements proposed assess changeability decay because they evaluate if changing a method requires more effort by measuring the amount of changes required, and the

complexity of such changes in particular periods of the method's lifetime.

2. Inaccurate instruments: It occurs when the measurements cannot determine uniquely the effects analyzed.

This remains to be one of the limitations of the methodology. Although it is likely that analysis phases point out inconsistencies and, therefore, possible bugs in the tools that perform the analysis, there is no way to ensure that the data collection is correct.

Gathering data in a different way for different groups of subjects: It occurs when the measurements are calibrated in between groups of subjects.

The tools used to analyze the applications are the same for all applications; therefore, there is no need to introduce any calibration in the data.

→ *Issues related to the researcher*

Also called expectancy bias, it refers to interpretations that favor the hypotheses.

1. Observer bias: It occurs when the researcher favors data that fulfils their expectations.

We think that the observer bias is minimal because most of the analysis is done automatically.

2. Interpreter bias: It occurs when the researcher favors interpretations that confirm their hypothesis.

Given that all evidence is considered and that the possible reasons for each piece of evidence are analyzed, the interpreter bias should be low.

3. Intentional bias: It occurs when the researcher modifies data to support his hypothesis. To minimize these issues it is recommended to replicate the experiment. The only requirement to replicate an experiment is to keep the same hypotheses.

All the hypotheses were obtained from the analysis of the literature (i.e. context phase 2). Moreover, the experiment has been replicated with different subject applications. Given that the results are similar regardless of the application analyzed, the chance of intentional bias should be low.

11.3.2 External validity

External validity says to what extent the results can be generalized across variations of set-up, subjects, contexts, etc. Therefore, these issues occur when the data selected to test the

hypotheses do not resemble “real life” data.

We think that, in general, the methodology and its case study on clones at the level of methods are based on real Open Source data. We do not know to what extent the applications chosen resemble industrial applications or large applications. However, given that the results were similar regardless of the application analyzed, we think that the results are valid for small-to-medium size, open source Java applications.

1. Selection of particular subjects: It occurs when subjects analyzed are special in any way. Given that they might not represent typical subjects, the results cannot be generalized.

The applications analyzed were selected under pre-established criteria: being written in Java, providing their repository, and having a repository that uses CVS. Besides, since the first step of the methodology (i.e. description of the applications to analyze) it is established that the results only apply to small-to-medium size, open source, Java applications.

2. Selection of the scale of the experiment: It occurs when the amount of subjects analyzed is not enough to generalize.

For the analyses related with changeability, we analyzed 5 systems through 9373 methods over 11623 commit transactions. For the analyses related only with cloned methods, regardless of the amount of changes of the method, we analyzed 16039 methods. Although, we think that this is a large dataset, it is difficult to know if it is enough to generalize to all small-to-medium size, open-source, Java applications. Nevertheless, given the consistency of the results obtained regardless the application analyzed, we consider that the scale was appropriate.

3. Selection of variables to control: It occurs when the set-up of the experiment is so restricted that it cannot reflect the concepts to study.

We think that the minimum size of the clone, the strategy to discard false positives (see section 5.1.1), and the strategy used to calculate the thresholds of the characteristics and the metrics when analyzing relative risk have affected the results.

11.4 Future work

This section summarizes some of the ways in which we envision this work could be continued. The section is organized in two sub-sections; one that discusses the perspectives regarding the

methodology, and another that discusses the possible continuations regarding the analysis of clones.

11.4.1 Methodology improvements and extensions

Work in the methodology can be extended in two ways: by improving problematic aspects of the current methodology, and by adapting the methodology to assess the effect of other implementation characteristics.

11.4.1.1 Improvements

During the adaptation and application of the methodology to the study of clones at method-level, we found some phases had could be improved. Most of the improvements proposed below are minor; however, we think that repeating the analysis with these improvements may result in stronger results:

1. Definition of the SCIUS: We think that it is better to gather all possible instances of the SCIUS because, in that way, interesting results about SCIUS that could be considered false positives are not excluded from the beginning.
2. Data collection: We used several third party tools to gather the data; however, the number of tools used can be reduced by exploiting better the capabilities of some tools. For instance, the output of SPOON could have been used to eliminate the need for CTAGS; some of its source code analysis functionality could have permitted storing richer information in the databases e.g. it could have helped to identify precise information about the type of changes that an application undergoes. Reducing the number of tools used would improve the quality of the data collection algorithms by reducing their complexity and improving their performance using all the information that third party tools provide.
3. Description of the applications to analyze: It might be difficult and even subjective deciding when several applications have a similar value for a characteristic.
4. Nature of the SCIUS: In terms of SCIUS nature, we have identified two drawbacks: First, given that cloned methods are analyzed one characteristic at a time, it is impossible to check whether interesting methods with respect to one characteristic are also interesting with respect to another one. Second, certain SCIUS define relations that

would enhance the analysis of their characteristics; e.g., in the case of clones, the analysis on names of the methods that contain them could take into account the families that relate them.

5. Evolution of the SCIUS: The analysis of SCIUS per package requires a lot of processing but its value is very limited. Unless the applications analyzed lay in the same domain it is very little that can be concluded from these analyses.

6. Effect of the SCIUS in changeability:

    a. Comparing the changeability metrics: The methodology assumed that the methods Always with the SCIUS (AI) and those Never with the SCIUS (NI) would have the same relation than the methods Sometimes with the SCIUS when having the SCIUS (SIwI), and when not having the SCIUS (SIwnI). However, this is might not be the case, therefore one should compare the changeability of all types of methods together, and leave the comparison AI vs. NI, and SIwI vs. SIwnI only for the statistical analyses that say if the distributions are different or not.

    b. Identification of characteristics that affect changeability: The definition of thresholds is not optimal to find large values of the characteristic or metric analyzed. This is because it is likely that the values analyzed do not follow a normal distribution. Take for instance the values of the size of the clone. Based on the analysis of the nature of clones we think that an appropriate value to decide whether or not the size of the clone is large is 100 tokens given that the minority of clones are above 100 tokens. However, our strategy based on finding the median 75 percentile gave us a threshold of 67 tokens. We think that a better way to find the thresholds is to follow the way in which box-plot graphs outliers. That is, finding the maximum value that lays inside the inter-quartile range. When doing this for the size of the clone we obtain a threshold of 107 tokens, which is a more appropriate threshold for high values, in fact is very close to the intuitive threshold obtained by analyzing the nature of clones.

    c. Classification of SCEs by changeability measurements: The usefulness of this phase is unknown. The output of this phase could be evaluated by predicting the changeability of methods of a new application based on its characteristics, and evaluate the accuracy of the prediction.

7. Relation of the SCIUS with other SCIs: the selection of the other SCIs should be decided based on hypothetical relations with the SCIUS.

11.4.1.2 Extensions

We think that the first extension that the methodology should have is including metrics to analyze quality attributes other than changeability. This change makes of the quality attribute analyzed a first class entity on the methodology, at the level of SCIs and SCEs, increasing the potential of the methodology to be applied in other research problems.

Other possible extensions of the methodology include analyzing to what extent Source Code Characteristics that are considered beneficial, such as reuse and variability maximization using product lines, indeed affect the changeability of the applications.

11.4.2 Analysis of clones at the level of methods

It is important to test the validity of the results obtained in other contexts such as closed-source applications, larger applications, and applications written in other programming languages, as well as other programming paradigms.

It would be also good to restructure these applications in such a way that it is easy to add modules to analyze different SCIs, as well as, applications using SCM systems other than CVS. In fact, the migration from CVS to SVN is an imperative because most of the open-source community has been migrating to SVN given that it implements natively key concepts such as atomic commit transactions.

Finally, there are several hypotheses originated from the analysis of clones at method level that require further investigation:

- If the percentage of method cloned in methods with exact clones decays slower than in methods with clones that are not exact.
- If SC-methods change only with their family when cloned.
- If SC-methods have clones that can be related to problems with the design or to lack of modularization mechanisms in the programming language.
- If methods with clones and feature envy indeed have higher changeability metrics.
- Find common characteristics among SC-methods.

11.5 Contributions

Three key insights are derived from our work:

The evidence shown in this thesis indicates that, according to the intuition of the academic community, the worse clones are those that do not change in the same way than the rest of their families. *We have identified these kinds of methods as SC-methods when not cloned, thereby confirming that once a cloned method diverges from the family of its clones; its changeability is greatly degraded (see conclusion 9 of section 11.2)*. Given that, SC-methods represent roughly half of the clones; this finding implies that if maintenance tasks focus on tackling clones indiscriminately, half of the resources will be wasted on eliminating harmless clones.

We have found that the effect of cloning in changeability is distinguished more by characteristics of the method than by characteristics of the clone (see conclusion 10 of section 11.2). This means that methods with certain characteristics (high fan-out and high LOC) when they acquire a clone, their changeability decay is accelerated; making this type of methods high-risk.

We also found that most of the changes in cloned methods occur in their cloned fragments (see conclusion 7 of section 11.2). This confirms that the increase of changeability decay on cloned methods is due to their cloned fragments.

Finally, we have shown that contrary to previous findings, cloned concepts are persistent (see conclusion 10 of section 11.2). Cloned fragments inside a method disappear from their family in a short amount of time. However, they are not refactored. In fact, they are changed consistently with other cloned fragments forming a new clone family. Therefore, although cloned fragments are volatile with respect to clone families, they are persistent with respect to the methods that host them.

We think that these results could help in prioritizing anti-regressive work because they offer a

characterization of cloned methods that are harmful i.e. those that are sometimes cloned, and have high fan-out and high LOCs. In this way, the results would not only offer some insight about the harmfulness of cloning, but also would contribute to improve the practitioners' work

References

[Adar '07] Adar, E. and M. Kim (2007). SoftGUESS: Visualization and Exploration of Code Clones in Context. Proc. Int'l Conf. on Software Engineering (ICSE)**:** 762-766.

[Aho '74] Aho, A. V. and S. C. Johnson (1974). "LR Parsing." ACM Comput. Surv. **6**(2): 99-124.

[Al-Ekram '05] Al-Ekram, R., C. J. Kapser, R. C. Holt and M. W. Godfrey (2005). Cloning by accident: an empirical study of source code cloning across software systems. Proc. of the Int'l Symp. on Empirical Software Engineering (ISESE)**:** 376-385.

[Antoniol '02] Antoniol, G., U. Villano, E. Merlo and M. Di Penta (2002). "Analyzing cloning evolution in the Linux kernel." Information and Software Technology **44**(13): 755-765.

[Antoniol '04] Antoniol, G., M. Di Penta and E. Merlo (2004). An automatic approach to identify class evolution discontinuities. Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)**:** 31-40.

[Arisholm '00] Arisholm, E. and D. I. K. Sjoberg (2000). "Towards a framework for empirical assessment of changeability decay." J. Syst. Softw. **53**(1): 3-14.

[Aversano '07] Aversano, L., L. Cerulo and M. D. Penta (2007). How Clones are Maintained: An Empirical Study. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 81-90.

[Baker '95] Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 86-95.

[Baker '07] Baker, B. S. (2007). "Finding Clones with Dup: Analysis of an Experiment." IEEE Trans. Softw. Eng. **33**(9): 608-621.

[Bakota '07] Bakota, T., R. Ferenc and T. Gyimothy (2007). Clone Smells in Software Evolution. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 24-33.

[Balazinska '00] Balazinska, M., E. Merlo, M. Dagenais, B. Lagüe and K. Kontogiannis (2000). Advanced Clone-Analysis to Support Object-Oriented System Refactoring. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 98-107.

[Balint '06] Balint, M., R. Marinescu and T. Girba (2006). How Developers Copy. Proc. of the Int'l

Conf. on Program Comprehension (ICPC), IEEE Computer Society**:** 56-68.

[Bandi '03] Bandi, R. K., V. K. Vaishnavi and D. E. Turk (2003). "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics." IEEE Trans. Softw. Eng. **29**(1): 77-87.

[Bär '99] Bär, H., M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A.-M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar and J. Weisbrod (1999). The FAMOOS Object-Oriented Reengineering Handbook.

[Basili '84] Basili, V. R. and B. T. Perricone (1984). "Software errors and complexity: an empirical investigation." **Commun. ACM** **27**(1): 42-52.

[Basili '96] Basili, V. R., L. C. Briand and W. L. Melo (1996). "A validation of object-oriented design metrics as quality indicators." IEEE Trans. Softw. Eng. **22**(10): 751-761.

[Basit '05a] Basit, H. A. and S. Jarzabek (2005a). Detecting higher-level similarity patterns in programs. Proc. of the European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng. (ESEC-FSE). Lisbon, Portugal, ACM Press**:** 156-165.

[Basit '05b] Basit, H. A., D. C. Rajapakse and S. Jarzabek (2005b). Beyond templates: a study of clones in the STL and some general implications. Proc. Int'l Conf. on Software Engineering (ICSE). St. Louis, MO, USA, ACM**:** 451-459.

[Baxter '98] Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna and L. Bier (1998). Clone Detection Using Abstract Syntax Trees. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 368-377.

[Belle '04] Belle, T. B. V. (2004). Modularity and the Evolution of Software Evolvability. Albuquerque, USA, The University of New Mexico.

[Bellon '07] Bellon, S., R. Koschke, G. Antoniol, J. A.-K. Krinke, J. and E. A.-M. Merlo, E. (2007). "Comparison and Evaluation of Clone Detection Tools." IEEE Trans. Softw. Eng. **33**(9): 577-591.

[Bennett '93] Bennett, B. and C. P. Satterthwaite (1993). A maintainability measure of embedded software. Proc. of the IEEE National Aerospace and Electronics Conf (NAECON)**:** 560-565 vol.1.

[Bennett '00] Bennett, K. H. and V. c. T. Rajlich (2000). Software maintenance and evolution: a roadmap. Proc. Conf. on The Future of Software Engineering. Limerick, Ireland, ACM Press**:** 73-87.

[Bianchi '01] Bianchi, A., D. Caivano, F. Lanubile and G. Visaggio (2001). Evaluating software degradation through entropy. Proc. of the int'l symp. on Software Metrics (METRICS), IEEE Computer Society**:** 210-219.

[Bieman '03] Bieman, J. M., A. A. Andrews and H. J. Yang (2003). Understanding Change-Proneness in OO Software through Visualization. Proc. Int'l Workshop on Program Comprehension (IWPC), IEEE Computer Society**:** 44-53.

[Boogerd '08] Boogerd, C. and L. Moonen (2008). Assessing the Value of Coding Standards: An Empirical Study. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 277-286.

[Brown '98] Brown, W., R. Malveau and T. Mowbray (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley.

[Burd '99] Burd, E. and M. Munro (1999). An Initial Approach towards Measuring and Characterizing Software Evolution. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 168-174.

[Burd '02] Burd, E. and J. Bailey (2002). Evaluating Clone Detection Tools for Use during Preventative Maintenance. Proc. of the int'l workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society**:** 36-43.

[Calefato '04] Calefato, F., F. Lanubile and T. Mallardo (2004). "Function Clone Detection in Web Applications: A Semiautomated Approach." Journal of Web Engineering **3**(1): 3-21.

[Capiluppi '04a] Capiluppi, A., M. Morisio and P. Lago (2004a). Evolution of understanding in OSS projects. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 58-68.

[Capiluppi '04b] Capiluppi, A., M. Morisio and J. F. Ramil (2004b). Structural evolution of an Open Source System: a case study. Proc. Int'l Workshop on Program Comprehension (IWPC), IEEE Computer Society**:** 172-182.

[Chou '01] Chou, A., J. Yang, B. Chelf, S. Hallem and D. Engler (2001). An empirical study of operating systems errors. Proc. of the eighteenth ACM symp. on Operating systems principles. Banff, Alberta, Canada, ACM**:** 73-88.

[Ciupke '99] Ciupke, O. (1999). Automatic Detection of Design Problems in Object-Oriented Reengineering. Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS), IEEE Computer Society**:** 18.

[Coad '91] Coad, P. and E. Yourdon (1991). Object-Oriented Design, Prentice Hall.

[Coleman '94] Coleman, D., D. Ash, B. Lowther and P. Oman (1994). "Using Metrics to Evaluate Software System Maintainability." Computer **27**(8): 44-49.

[Cordy '03] Cordy, J. R. (2003). Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. Proc. Int'l Workshop on Program Comprehension (IWPC) as keynote paper. Portland, Oregon**:** 196-206.

[Cordy '04] Cordy, J. R., T. R. Dean and N. Synytskyy (2004). Practical language-independent detection of near-miss clones. Proc. of the conf. of the Centre for Advanced Studies on Collaborative research (CASCON). Markham, Ontario, Canada, IBM Press**:** 1-12.

[Crespo '05] Crespo, Y., C. López, R. Marticorena and E. Manso (2005). Language Independent Metrics Support towards Refactoring Inference. Proc. of the Int'l workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE). Glasgow, UK**:** 18-29.

[D'Ambros '08] D'Ambros, M., H. C. Gall, M. Lanza and M. Pinzger (2008). Analyzing software

repositories to understand software evolution. Software Evolution. T. Mens and S. Demeyer, Springer**:** 39-69.

[Dagpinar '03] Dagpinar, M. and J. H. Jahnke (2003). Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 155-164.

[Darcy '05] Darcy, D. P., C. F. Kemerer, S. A. Slaughter and J. E. Tomayko (2005). "The Structural Complexity of Software: An Experimental Test." IEEE Trans. Softw. Eng. **31**(11): 982-995.

[Davey '00] Davey, J. and E. Burd (2000). Evaluating the Suitability of Data Clustering for Software Remodularization. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 268-276.

[Davey '95] Davey, N., P. Barson, S. Field, R. Frank and S. Tansley (1995). "The Development of a Software Clone Detector." International Journal of Applied Software Technology **1**(3): 219-236.

[Demeyer '00] Demeyer, S., S. Ducasse and O. Nierstrasz (2000). Finding refactorings via change metrics. Proc. int'l conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA). Minneapolis, Minnesota, United States, ACM Press**:** 166-177.

[Dijkstra '82] Dijkstra, E. W. (1982). On the role of scientific thought. Selected writings on Computing: A Personal Perspective. New York, NY, USA, Springer-Verlag**:** 60–66.

[Ducasse '99] Ducasse, S., M. Rieger and S. Demeyer (1999). A Language Independent Approach for Detecting Duplicated Code. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 109-118.

[Ducasse '06] Ducasse, S., O. Nierstrasz and M. Rieger (2006). "On the effectiveness of clone detection by string matching." J. Softw. Maint. Evol. **18**(1): 37-58.

[Ducasse '07] Ducasse, S., D. Pollet, M. Suen, H. Abdeen and I. Alloui (2007). Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 94-103.

[Eick '01] Eick, S. G., T. L. Graves, A. F. Karr, J. S. Marron and A. Mockus (2001). "Does Code Decay? Assessing the Evidence from Change Management Data." IEEE Trans. Softw. Eng. **27**(1): 1-12.

[Emden '02] Emden, E. V. and L. Moonen (2002). Java Quality Assurance by Detecting Code Smells. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 97-106.

[Evans '07] Evans, W. S., C. W. Fraser and F. Ma (2007). Clone Detection via Structural Abstraction. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 150-159.

[Fenton '00] Fenton, N. E. and N. Ohlsson (2000). "Quantitative Analysis of Faults and Failures in a Complex Software System." IEEE Trans. Softw. Eng. **26**(8): 797-814.

[Ferneley '99] Ferneley, E. H. (1999). "Design metrics as an aid to software maintenance: an empirical study." J. Softw. Maint. Evol. **11**(1): 55-72.

[Fioravanti '01] Fioravanti, F. and P. Nesi (2001). "Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems." IEEE Trans. Softw. Eng. **27**(12): 1062-1084.

[Firesmith '03] Firesmith, D. G. (2003). Common concepts underlying safety, security, and survivability engineering. Technical Note CMU/SEI-2003-TN-033. C. M. S. E. Institute. http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tn033.pdf.

[Fowler '99] Fowler, M., K. Beck, J. Brant, W. Opdyke and D. Roberts (1999). Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional.

[Gaffney '81] Gaffney, J. E. (1981). Metrics in software quality assurance. ACM conference, ACM**:** 126-130.

[Gall '97] Gall, H., M. Jazayeri, R. e. Klosch and G. Trausmuth (1997). Software evolution observations based on product release history. Proc. Int'l Conf. on Software Maintenance (ICSM). Bari, Italy, IEEE Computer Society**:** 160-166.

[Gall '98] Gall, H., K. Hajek and M. Jazayeri (1998). Detection of logical coupling based on product release history. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 190-198.

[Geiger '06] Geiger, R., B. Fluri, H. C. Gall and M. Pinzger (2006). Relation of Code Clones and Change Couplings. Proc. of the Int'l Conf. of Fundamental Approaches to Software Engineering (FASE), Springer**:** 411-425.

[German '04] German, D. M. (2004). An Empirical Study of Fine-Grained Software Modifications. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 316-325.

[Giesecke '07] Giesecke, S. (2007). Generic modeling of code clones. Duplication, Redundancy, and Similarity in Software. R. Koschke, E. Merlo and A. Walenstein, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[Girba '04a] Girba, T., S. Ducasse and M. Lanza (2004a). Yesterday's Weather: guiding early reverse engineering efforts by summarizing the evolution of changes. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 40-49.

[Girba '04b] Girba, T., S. Ducasse, R. Marinescu and D. Ratiu (2004b). Identifying Entities That Change Together. Workshop on Empirical Studies of Software Maintenance (WESS).

[Girba '05] Girba, T., M. Lanza and S. Ducasse (2005). Characterizing the Evolution of Class Hierarchies. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 2-11.

[Godfrey '02] Godfrey, M. and Q. Tu (2002). Tracking structural evolution using origin analysis. Proc. Int'l Workshop on Principles of Software Evolution (IWPSE). Orlando, Florida, ACM Press**:** 117-119.

[Godfrey '00] Godfrey, M. W. and Q. Tu (2000). Evolution in Open Source Software: A Case Study. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 131-142.

[Godfrey '05] Godfrey, M. W. and L. Zou (2005). "Using Origin Analysis to Detect Merging and Splitting

of Source Code Entities." IEEE Trans. Softw. Eng. **31**(2): 166-181.

[Golomingi '01] Golomingi, G. (2001). A scenario based approach for refactoring duplicated code in object oriented systems. Diploma Thesis, University of Bern.

[Gosling '96] Gosling, J., B. Joy and G. L. Steele (1996). The Java Language Specification, Addison-Wesley Longman Publishing Co., Inc.

[Gosling '00] Gosling, J., B. Joy, G. L. Steele and G. Bracha (2000). The Java Language Specification, Prentice Hall PTR.

[Gosling '05] Gosling, J., B. Joy, G. L. Steele and G. Bracha (2005). The Java Language Specification, Addison Wesley.

[Graves '98] Graves, T. L. and A. Mockus (1998). Inferring Change Effort from Configuration Management Databases. Proc. int'l symp. on Software Metrics (METRICS), IEEE Computer Society**:** 267-273.

[Green '00] Green, M., M. Freedman and L. Gordis (2000). Reference Guide on Epidemiology. Reference Manual on Scientific Evidence, Federal Judicial Center: Washington, DC**:** 638.

[Griswold '93] Griswold, W. G. and D. Notkin (1993). "Automated Assistance for Program Restructuring." ACM Trans. Softw. Eng. Methodol. **2**(3): 228-269.

[Grottke '06] Grottke, M., L. Li, K. Vaidyanathan and K. S. Trivedi (2006). "Analysis of software aging in a web server." IEEE Transactions on Reliability **55**(3): 411-420.

[Halstead '77]Halstead, M. H. (1977). Elements of Software Science (Operating and programming systems series), Elsevier Science Inc.

[Harrison '98]Harrison, R., S. J. Counsell and R. V. Nithi (1998). "An Evaluation of the MOOD Set of Object-Oriented Software Metrics." IEEE Trans. Softw. Eng. **24**(6): 491-496.

[Hassan '04] Hassan, A. E. and R. C. Holt (2004). Predicting change propagation in software systems. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 284-293.

[Herraiz '07] Herraiz, I., J. M. Gonzalez-Barahona and G. Robles (2007). Towards a Theoretical Model for Software Growth. Proc. of the Int'l Workshop on Mining Software Repositories (MSR). J. M. Gonzalez-Barahona, IEEE Computer Society**:** 21-28.

[IEEE '99] IEEE (1999). Standard Glossary of Software Engineering Terminology 610.12-1990. IEEE Standards Software Engineering, IEEE Press. **one: Customer and Terminology Standards**.

[ISO '01] ISO (2001). ISO/IEC 9126 -1, Software Engineering – Product Quality – Part 1: Quality model. Geneva, International Organization for Standardization**:** 46.

[Jarzabek '06]Jarzabek, S. and S. Li (2006). "Unifying clones with a generative programming technique: a case study." J. Softw. Maint. Evol. **18**(4): 267-292.

[Jiang '07] Jiang, L., Z. Su and E. Chiu (2007). Context-based detection of clone-related bugs. Proc. of the European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng. (ESEC-FSE).

Dubrovnik, Croatia, ACM**:** 55-64.

[Johnson '93] Johnson, J. H. (1993). Identifying redundancy in source code using fingerprints. Proc. of the conf. of the Centre for Advanced Studies on Collaborative research (CASCON). Toronto, Ontario, Canada, IBM Press**:** 171-183.

[Johnson '94] Johnson, J. H. (1994). Substring Matching for Clone Detection and Change Tracking. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 120-126.

[Johnson '88] Johnson, R. E. and B. Foote (1988). "Designing Reusable Classes." Journal of Object-Oriented Programming **1**(2): 22-35.

[Kamiya '02] Kamiya, T., S. Kusumoto and K. Inoue (2002). "CCFinder: a multilinguistic token-based code clone detection system for large scale source code." IEEE Trans. Softw. Eng. **28**(7): 654-670.

[Kapser '03] Kapser, C. and M. Godfrey (2003). Toward a Taxonomy of Clones in Source Code: A Case Study. Proc. of the int'l workshop on Evolution of Large-scale Industrial Software Applications (ELISA)**:** 67-78.

[Kapser '04] Kapser, C. and M. Godfrey (2004). Aiding Comprehension of Cloning Through Categorization. Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), IEEE Computer Society**:** 85-94.

[Kapser '05] Kapser, C. and M. Godfrey (2005). Improved Tool Support for the Investigation of Duplication in Software. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 305-314.

[Kapser '06a] Kapser, C. and M. Godfrey (2006a). 'Cloning considered harmful' considered harmful. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 19-28.

[Kapser '06b] Kapser, C. and M. Godfrey (2006b). "Supporting the analysis of clones in software systems." J. Softw. Maint. Evol. **18**(2): 61-82.

[Kapser '07] Kapser, C., P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. v. Rysselberghe and P. Weißgerber (2007). Subjectivity in Clone Judgment: Can We Ever Agree? Duplication, Redundancy, and Similarity in Software. R. Koschke, E. Merlo and A. Walenstein, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[Kapser '08] Kapser, C. J. and M. W. Godfrey (2008). "'Cloning considered harmful' considered harmful: patterns of cloning in software." Empirical Softw. Engg. **13**(6): 645-692.

[Kataoka '01] Kataoka, Y., M. Ernst, W. Griswold and D. Notkin (2001). Automated Support for Program Refactoring using Invariants. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 736-743.

[Keeffe '04] Keeffe, M. Ó. and M. Ó. Cinnéide (2004). Towards Automated Design Improvement Through Combinatorial Optimisation. Proc. of the workshop on Directions in Software Engineering Environments (WoDiSEE). Edinburgh**:** 75-82.

[Kim '04] Kim, M., L. Bergman, T. Lau and D. Notkin (2004). An ethnographic study of copy and

paste programming practices in OOPL. Proc. of the Int'l Symp. on Empirical Software Engineering (ISESE)**:** 83-92.

[Kim '05] Kim, M., V. Sazawal, D. Notkin and G. Murphy (2005). An empirical study of code clone genealogies. Proc. of the European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng. (ESEC-FSE), ACM Press**:** 187-196.

[Kitchenham '96] Kitchenham, B. and S. L. Pfleeger (1996). "Software quality: the elusive target." Software, IEEE **13**(1): 12-21.

[Komondoor '03] Komondoor, R. and S. Horwitz (2003). Effective, Automatic Procedure Extraction. Proc. Int'l Workshop on Program Comprehension (IWPC), IEEE Computer Society**:** 33.

[Kontogiannis '97] Kontogiannis, K. (1997). Evaluation experiments on the detection of programming patterns using software metrics. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 44-54.

[Kontogiannis '96] Kontogiannis, K. A., R. Demori, E. Merlo, M. Galler and M. Bernstein (1996). Pattern matching for clone and concept detection. Reverse engineering, Kluwer Academic Publishers**:** 77-108.

[Koschke '06]Koschke, R., R. Falke and P. Frenzel (2006). Clone Detection Using Abstract Syntax Suffix Trees. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 253-262.

[Koschke '07]Koschke, R. (2007). Survey of Research on Software Clones. Duplication, Redundancy, and Similarity in Software. R. Koschke, E. Merlo and A. Walenstein, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[Krinke '01] Krinke, J. (2001). Identifying Similar Code with Program Dependence Graphs. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 301-309.

[Krinke '07] Krinke, J. (2007). A Study of Consistent and Inconsistent Changes to Code Clones. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society.

[Krinke '08] Krinke, J. (2008). Is Cloned Code more stable than Non-Cloned Code? Proc. of the int'l workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society**:** 57-66.

[Lague '97] Lague, B., D. Proulx, J. Mayrand, E. M. Merlo and J. Hudepohl (1997). Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 314-321.

[Lakos '96] Lakos, J. (1996). Large-Scale C++ Software Design, Addison-Wesley Professional.

[Lehman '85] Lehman, M. M. and L. A. Belady (1985). Program Evolution: Processes of Software Change. London, Academic Press.

[Lehman '97] Lehman, M. M., J. F. Ramil, P. D. Wernick, D. E. Perry and W. M. Turski (1997). Metrics and laws of software evolution-the nineties view. Proc. Int'l Software Metrics Symp. (METRICS). J. F. Ramil**:** 20-32.

[Li '06] Li, Z., S. Lu, S. Myagmar and Y. Zhou (2006). "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code." IEEE Trans. Softw. Eng. **32**(3): 176-192.

[Lieberherr '88] Lieberherr, K. J., I. Holland and A. Riel (1988). Object-Oriented Programming: An Objective Sense of Style. Proc. int'l conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)**:** 323-334.

[Lientz '81] Lientz, B. P. and E. B. Swanson (1981). "Problems in application software maintenance." Commun. ACM **24**(11): 763-769.

[Liskov '87] Liskov, B. (1987). Data abstraction and hierarchy. Proc. int'l conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) as keynote paper. Orlando, Florida, United States, ACM**:** 17-34.

[Liu '06] Liu, C., C. Chen, J. Han and P. S. Yu (2006). GPLAG: detection of software plagiarism by program dependence graph analysis. Proc. int'l conf. on Knowledge Discovery and Data mining. Philadelphia, PA, USA, ACM**:** 872-881.

[Livshits '05] Livshits, B. and T. Zimmermann (2005). DynaMine: finding common error patterns by mining software revision histories. Proc. of the European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng. (ESEC-FSE). Lisbon, Portugal, ACM Press**:** 296-305.

[Lozano '06] Lozano, A., M. Wermelinger and B. Nuseibeh (2006). Degradation archaeology: studying software flaws' evolution (Position paper). Proc. Int'l Workshop on Software Evolution (Evol)**:** 119 - 126.

[Lozano '07a] Lozano, A., M. Wermelinger and B. Nuseibeh (2007a). Evaluating the harmfulness of cloning: a change based experiment. Proc. of the int'l workshop on Mining Software Repositories (MSR), IEEE Computer Society**:** 18-21.

[Lozano '07b] Lozano, A., M. Wermelinger and B. Nuseibeh (2007b). Assessing the impact of bad smells using historical information (Position paper). Proc. of the Int'l Workshop On Principles of Software Evolution (IWPSE), IEEE Computer Society.

[Lozano '08a] Lozano, A. (2008a). A Methodology to Assess the Impact of Source Code Flaws in Changeability, and its Application to Clones (doctoral symposium). Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society.

[Lozano '08b] Lozano, A. and M. Wermelinger (2008b). Assessing the effect of clones on changeability. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 227-236.

[Lozano '08c] Lozano, A., M. Wermelinger and B. Nuseibeh (2008c). Evaluating the relation between changeability decay and the characteristics of clones and methods. Proc. Int'l Workshop on Software Evolution (Evol)**:** 100-109.

[Lucca '02] Lucca, G. A. D., M. D. Penta and A. R. Fasolino (2002). An Approach to Identify Duplicated Web Pages. Proc. Int'l Computer Software and Applications Conf. (COMPSAC) on Prolonging Software Life: Development and Redevelopment, IEEE Computer Society**:** 481-486.

[Marcus '01] Marcus, A. and J. I. Maletic (2001). Identification of High-Level Concept Clones in Source Code. Proc. of the Int'l Conf. on Automated Software Engineering (ASE), IEEE Computer

Society**:** 107-114.

[Marinescu '01] Marinescu, R. (2001). Detecting design flaws via metrics in Object Oriented Systems. Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS)**:** 173-182.

[Marinescu '02] Marinescu, R. (2002). Measurement and Quality in Object-Oriented Design, Politehnica University of Timisoara.

[Marinescu '04a] Marinescu, R. (2004a). Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 350-359.

[Marinescu '04b] Marinescu, R. and D. Ratiu (2004b). Quantifying the Quality of Object-Oriented Design: The Factor-Strategy Model. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 192-201.

[Martin '96a] Martin, R. C. (1996a). The Dependency Inversion Principle. The C++ Report.

[Martin '96b] Martin, R. C. (1996b). Granularity. The C++ Report.

[Martin '96c] Martin, R. C. (1996c). Stability. The C++ Report.

[Martin '00] Martin, R. C. (2000). Design Principles and Design Patterns. http://www.objectmentor.com.

[Mayrand '96] Mayrand, J., C. Leblanc and E. Merlo (1996). Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 244-253.

[McCabe '76] McCabe, T. J. (1976). "A complexity measure." IEEE Trans. Softw. Eng **2**(4): 308- 320.

[Mens '03] Mens, T., T. Tourwé and F. Muñoz (2003). Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring. Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), IEEE Computer Society**:** 39-44.

[Merlo '07] Merlo, E. (2007). Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity. Duplication, Redundancy, and Similarity in Software. R. Koschke, E. Merlo and A. Walenstein, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[Meyer '92] Meyer, B. (1992). "Applying `design by contract'." Computer **25**(10): 40-51.

[Mockus '00] Mockus, A. and D. M. Weiss (2000). "Predicting risk of software changes." Bell Labs Technical Journal **5**(2): 169-180.

[Moha '06a] Moha, N., Y.-G. Guéhéneuc and P. Leduc (2006a). Automatic Generation of Detection Algorithms for Design Defects. Proc. of the Int'l Conf. on Automated Software Engineering (ASE), IEEE Computer Society**:** 297-300.

[Moha '06b] Moha, N., D.-L. Huynh and Y.-G. Guéhéneuc (2006b). Une taxonomie et un métamodèle pour la détection des défauts de conception. Actes du colloque Langages et Modèles à Objets**:** 201-216.

[Moha '08] Moha, N. (2008). DECOR : Détection et correction des défauts dans les systèmes orientés objet. Montréal, Canada, Université de Montréal.

[Monden '02] Monden, A., D. Nakae, T. Kamiya, S.-i. Sato and K.-i. Matsumoto (2002). Software Quality Analysis by Code Clones in Industrial Legacy Software. Proc. int'l symp. on Software Metrics (METRICS), IEEE Computer Society**:** 87-94.

[Moody '05] Moody, D. L. (2005). "Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions." Data Knowl. Eng. **55**(3): 243-276.

[Moore '96] Moore, I. (1996). Automatic inheritance hierarchy restructuring and method refactoring. Proc. int'l conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA). San Jose, California, United States, ACM Press**:** 235-250.

[Munro '05] Munro, M. J. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. Proc. of the int'l symp. on Software Metrics (METRICS), IEEE Computer Society**:** 15-24.

[Murphy '96] Murphy, G. C. and D. Notkin (1996). "Lightweight lexical source model extraction." ACM Trans. Softw. Eng. Methodol. **5**(3): 262-292.

[Myers '03] Myers, C. R. (2003). "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs." Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **68**(4): 046116-15.

[Nance '88] Nance, R. E. and J. D. Arthur (1988). The methodology roles in the realization of a model development environment. Proc. 20th conf. on Winter simulation. San Diego, California, United States, ACM**:** 220-225.

[Oman '92] Oman, P. and J. Hagemeister (1992). Metrics for assessing a software system's maintainability. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 337-344.

[Parnas '72] Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules." Commun. ACM **15**(12): 1053-1058.

[Parnas '79] Parnas, D. L. (1979). "Designing software for ease of extension and contraction." IEEE Trans. Softw. Eng. **5**(2): 128-138.

[Parnas '94] Parnas, D. L. (1994). Software aging. Proc. Int'l Conf. on Software Engineering (ICSE). Sorrento, Italy, IEEE Computer Society Press**:** 279-287.

[Pawlak '06] Pawlak, R., C. Noguera and N. Petitprez (2006). Spoon: Program Analysis and Transformation in Java. INRIA Technical Report 5901.

[Peercy '81] Peercy, D. E. (1981). "A Software Maintainability Evaluation Methodology." IEEE Trans. Softw. Eng. **7**(4): 343-351.

[Ratiu '04] Ratiu, D., S. Ducasse, T. Girba and R. Marinescu (2004). Using History Information to Improve Design Flaws Detection. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 223-232.

[Riel '96] Riel, A. (1996). Object-Oriented Design Heuristics, Addison-Wesley Professional.

[Roy '08] Roy, C. K. and J. R. Cordy (2008). NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. Proc. of the Int'l Conf. on Program Comprehension (ICPC), IEEE Computer Society**:** 172-181.

[Rysselberghe '03] Rysselberghe, F. V. and S. Demeyer (2003). Reconstruction of Successful Software Evolution Using Clone Detection. Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), IEEE Computer Society**:** 126-130.

[Sanders '95] Sanders, J. and E. Curran (1995). Software Quality, A framework for success in software development and support, Addison-Wesley Professional.

[Seng '05] Seng, O., M. Bauer, M. Biehl and G. Pache (2005). Search-based improvement of subsystem decompositions. Proc. of the conf. on Genetic and Evolutionary Computation (GECCO). Washington DC, USA, ACM Press**:** 1045-1051.

[Simon '01] Simon, F., F. Steinbrückner and C. Lewerentz (2001). Metrics Based Refactoring. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 30-38.

[Singer '98] Singer, J. (1998). Practices of software maintenance. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 139-145.

[Smith '09] Smith, R. and S. Horwitz (2009). Detecting and Measuring Similarity in Code Clones. Proc. Int'l Workshop on Software Clones (IWSC).

[Subramanyam '03] Subramanyam, R. and M. S. Krishnan (2003). "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects." IEEE Trans. Softw. Eng. **29**(4): 297-310.

[Tahvildari '02] Tahvildari, L. and K. Kontogiannis (2002). A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 77-86.

[Tahvildari '03] Tahvildari, L. and K. Kontogiannis (2003). A Metric-Based Approach to Enhance Design Quality through Meta-pattern Transformations. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 183-192.

[Tahvildari '04] Tahvildari, L. and K. Kontogiannis (2004). Developing a Multi-Objective Decision Approach to Select Source-Code Improving Transformations. Proc. Int'l Conf. on Software Maintenance (ICSM), IEEE Computer Society**:** 427-431.

[Taubes '95] Taubes, G. and C. Mann (1995). Epidemiology faces its limits. Science**:** 164-169.

[Trifu '04] Trifu, A., O. Seng and T. Genssler (2004). Automated Design Flaw Correction in Object-Oriented Systems. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society**:** 174-183.

[Trifu '05] Trifu, A. and R. Marinescu (2005). Diagnosing Design Problems in Object Oriented Systems. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 155-164.

[Ueda '02] Ueda, Y., T. Kamiya, S. Kusumoto and K. Inoue (2002). Gemini: maintenance support environment based on code clone analysis. Proc. int'l symp. on Software Metrics (METRICS), IEEE Computer Society**:** 67-76.

[Wahler '04] Wahler, V., D. Seipel, J. W. v. Gudenberg and G. Fischer (2004). Clone Detection in Source Code by Frequent Itemset Techniques. Proc. of the int'l workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society**:** 128-135.

[Walenstein '04] Walenstein, A., A. Lakhotia and R. Koschke (2004). "The Second International Workshop on Detection of Software Clones: workshop report." SIGSOFT Softw. Eng. Notes **29**(2): 1-5.

[Walter '05] Walter, B. and B. Pietrzak (2005). Multi-criteria Detection of Bad Smells in Code with UTA Method. Extreme Programming and Agile Processes in Software Engineering, Lecture Notes in Computer Science**:** 154-161.

[Welker '95] Welker, K. D. and P. W. Oman (1995). "Software Maintainability Metrics Models in Practice." J. Defense Softw. Engin. **8**(11): 19-23.

[Wu '06] Wu, J. (2006). Open Source Software Evolution and Its Dynamics. Waterloo, Canada, University of Waterloo.

[Xing '04] Xing, Z. and E. Stroulia (2004). Understanding Class Evolution in Object-Oriented Software. Proc. Int'l Workshop on Program Comprehension (IWPC), IEEE Computer Society**:** 34-43.

[Xing '05] Xing, Z. and E. Stroulia (2005). UMLDiff: an algorithm for object-oriented design differencing. Proc. of the Int'l Conf. on Automated Software Engineering (ASE). Long Beach, CA, USA, ACM Press**:** 54-65.

[Yi '08] Yi, J. S., Y.-a. Kang, J. T. Stasko and J. A. Jacko (2008). Understanding and characterizing insights: how do people gain insights using information visualization? Proc. int'l conf. on BEyond time and errors: novel evaLuation methods for Information Visualization (BELIEV). Florence, Italy, ACM**:** 1-6.

[Ying '04] Ying, A. T. T., G. C. Murphy, R. Ng and M. C. Chu-Carroll (2004). "Predicting source code changes by mining change history." IEEE Trans. Softw. Eng. **30**(9): 574-586.

[Zimmermann '03] Zimmermann, T., S. Diehl and A. Zeller (2003). How history justifies system architecture (or not). Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)**:** 73-83.

[Zimmermann '04] Zimmermann, T. and P. Weibgerber (2004). Preprocessing CVS data for fine-grained analysis. Proc. of the int'l workshop on Mining Software Repositories (MSR)**:** 2-6.

[Zou '03] Zou, L. and M. W. Godfrey (2003). Detecting merging and splitting using origin analysis. Proc. Working Conf. on Reverse Engineering (WCRE), IEEE Computer Society**:** 146-154.

Appendix A Key concepts used in the methodology

This appendix presents a series of formulas that increase the level of accuracy of some concepts used across the methodology.

A.1 History concepts

This section of the appendix presents the formulas relevant to explain the concepts behind the analysis of the history of an application.

Source Code Entities (SCE): building units of an application. Most of the analysis and results that the methodology produce are at the level of SCEs. The set of entities are identified by an upper case E, particular entities are identified with a lower case letter e.

$$e : E$$

Each SCE is composed of one or several SCEs of the same or of lower granularity. For example, in Java, packages are composed of other packages or classes, classes being of a lower level of granularity than packages. Depending on the SCIUS, the pertinent granularity of the SCEs to be analyzed may vary. For instance, for analyzing god classes it is enough to have information of the properties at class level, while for analyzing brain methods it is necessary to have information of the properties at method level. The function sce gives the set of SCEs that compose an entity e.

$$sce : E \rightarrow set(E)$$
$$sce(e) = e2 \mid e2 \in e$$

Equation A 1. Source Code Entity (SCE)

Level: it is a number that indicates the granularity of a SCE e.

$$level : E \rightarrow N$$
$$level(e) = k \mid e \in E^k$$

Equation A 2. Level of a SCE

For more details see **Error! Not a valid bookmark self-reference.** that contains a suggestion

for granularity levels in Object Oriented languages. The level of granularity in which a formula operates would be, by default, the level of SCEs analyzed. In such cases, the level of granularity is not specified. In cases where the formula should be applied SCEs of granularity different to the level of SCEs analyzed, it would be explicit in the range of the formula. The level of the formula does not necessarily state one type of SCEs. If the level of the SCEs referred by the formula is higher than the one analyzed, a sign + would identify the level to which the formula applies i.e. E^+. In case the level of the SCEs referred by the formula is lower than the one analyzed, a sign - would identify the level to which the formula applies i.e. E^-.

Table A 1. Levels of granularity for SCEs[23]

Granularity Level	SCE
0	Characters
1	Tokens
2	Lines
3	Methods
4	Classes
5	Files
6	Modules/Packages

An alternative way to refer to the components of a SCE, is to refer to that SCE at a lower granularity level, for instance the lines of a method e can be referred as $sce(e)$ or as e^2.

The super-indices and the sub-indices have different semantics. The super-indices refer to the granularity of a SCE. The sub-indices refer to the order of a SCE. For instance, the term e_1^2 would refer to the first line of the method e.

[23] The methodology does not make sense when applied at certain levels of abstraction, such as characters or tokens. Such low levels of granularity are stated to be able to express more complex definitions used later. Examples of usages of these levels of abstraction can be found in the definition of clones, in the definition of the extension of the SCIUS inside SCEs, and in the definition of the stability in SCEs due to the SCIUS.

Snapshot: it is the set of entities that compose the application at a moment in time i. A snapshot describes the status of the application, in terms of the entities that compose it at a moment in time. There is a total order on the snapshots of the application; because of this, we model the snapshots as a function that maps consecutive ids to their corresponding set of source code entities. Note that this modeling excludes the reasoning on branches of the history of the application.

$$s : ℵ \rightarrow set(E)$$

Equation A 3. Snapshot

History of an application: is a sequence of snapshots in ascendant order by time.

$$h : seq(S)$$

Equation A 4. History of an application

Deleted entities: is the set of entities that were deleted at a point i in the history of the application. That is, the set of entities that belong only to the first of two consecutive snapshots i.e. s_{i-1}.

$$d : ℵ \rightarrow set(E)$$
$$d_i = s_{i-1} - s_i$$

Equation A 5. SCEs deleted in a logical change

Unchanged entities: is the set of entities that were not changed, nor deleted at a point i in the history of the application. That is, the set of entities that are repeated in two consecutive snapshots i.e. in s_{i-1} and s_i.

$$u : ℵ \rightarrow set(E)$$
$$u_i = s_i \cap s_{i-1}$$

Equation A 6. SCEs unchanged after a logical change

Changed entities or logical changes: is the set of entities that changed at a point i in the history of the application. That is, the set of entities that are not exact in the two snapshots, but the entity in the later snapshot $e2$ can be identified as a modified version of the entity in the early snapshot $e1$. The two versions of the same entity are matched by name, or by percentage of equal contents, i.e. if the SCEs that compose the entities resemble in at least 70%. Matching two entities that have a different name is called origin analysis. Several approaches to perform

origin analysis are explained in section 2.3.1, while the approach we propose is explained in section 7.2.2.1.

$$c : \aleph \rightarrow set(E)$$

$$c_i = \left\{ e1 \in (s_i - s_{i-1}) \;\middle|\; \left(\exists e2 \in (s_{i-1} - s_i) \;\middle|\; \begin{array}{l} name(e1) = name(e2) \vee \\ \\ \dfrac{\|sce(e1) \cap sce(e2)\|}{\min(\|sce(e1)\|, \|sce(e1)\|)} > 0.7 \end{array} \right) \right\}$$

Equation A 7. SCEs changed after a logical change

Commit transaction: is the set of entities that were changed, added, or deleted at a point i in the history of the application.

$$ct : \aleph \rightarrow set(E)$$
$$ct_i = s_i \setminus s_{i-1}$$

Equation A 8. Commit transaction

HasSCIUS: it is a predicate that states if a SCE e has the SCIUS at a given point i in the history of the application.

$$hasSCIUS : E \times \aleph \rightarrow Boolean$$
$$hasSCIUS(e, i) = \begin{cases} if\ e\ has\ the\ scius\ at\ i \Rightarrow True \\ otherwise \Rightarrow False \end{cases}$$

Equation A 9. Has the SCIUS

A.2 Clones

This section of the appendix presents the formulas relevant to explain the concepts behind the definition of clones.

A fragment of a method is a sequence of tokens. There are two types of tokens: syntax tokens and semantic tokens. A syntax token is a set of characters that has a meaning in the programming language. The syntax token are identified with a predicate that says if a token is either a restricted word or operator of the programming language.

$$syntax : E^1 \rightarrow Boolean$$
$$syntax(e) = \{e \in (KEYWORDS \cup OPERATORS)\}$$

Equation A 10. Syntax tokens

Semantic tokens are the complement of syntax tokens. This means that semantic tokens include literals, types, and calls to methods.

$$\mathbf{semantic} : \mathbf{E}^1 \rightarrow \mathbf{Boolean}$$
$$\mathbf{semantic}(\mathbf{e}) = \{\mathbf{e} \notin (\mathbf{KEYWORDS} \cup \mathbf{OPERATORS})\}$$

Equation A 11. Semantic tokens

Two fragments are similar if there is a correspondence in the type of their tokens. That is, for every syntax token in the method $\mathbf{e}$ there is a syntax token in the clone; and for every semantic token in the method $\mathbf{e}$ here is a semantic token in the clone. The value of syntax tokens should be identical, but the value of semantic tokens may change.

The function $\mathbf{similar\_fragment}$ returns the sequence of tokens that two methods have with the same type, and with the same value, in case of syntax tokens. Note that in the formula below the term $\mathbf{e}1^1_{\mathbf{j}+\mathbf{i}}$ refers to the $\mathbf{j+i^{th}}$ token of the method $\mathbf{e1}$.

$$\mathbf{similar\_fragment} : \mathbf{E}^3 \times \mathbf{E}^3 \rightarrow \mathbf{seq}(\mathbf{E}^1)$$
$$\mathbf{similar\_fragment}(\mathbf{e}1, \mathbf{e}2) = \bigcup_{\mathbf{i}} \mathbf{e}2^1_{\mathbf{j}+\mathbf{i}} \left| \exists \mathbf{j}, \mathbf{k} \left| \begin{array}{l} \left(\mathbf{syntax}\left(\mathbf{e}1^1_{\mathbf{j}+\mathbf{i}}\right) = \mathbf{syntax}\left(\mathbf{e}2^1_{\mathbf{k}+\mathbf{i}}\right)\right) \\ \wedge \\ \left(\mathbf{syntax}\left(\mathbf{e}1^1_{\mathbf{j}+\mathbf{i}}\right) \wedge \mathbf{syntax}\left(\mathbf{e}2^1_{\mathbf{k}+\mathbf{i}}\right) \wedge \mathbf{e}1^1_{\mathbf{j}+\mathbf{i}} = \mathbf{e}2^1_{\mathbf{k}+\mathbf{i}}\right) \end{array} \right. \right.$$

Equation A 12. Similarity of code fragments

The function $\mathbf{copy}$ returns the method $\mathbf{e2}$ with which the given method $\mathbf{e1}$ has the largest similar fragment. We call to the method $\mathbf{e2}$ the copy.

$$\mathbf{copy} : \mathbf{E}^3 \rightarrow \mathbf{E}^3$$
$$\mathbf{copy}(\mathbf{e}1) = \mathbf{e}2 \in \mathbf{E}^3 \left| \begin{array}{l} \forall \mathbf{e}3 \in \mathbf{E}^3 \\ \left\|\mathbf{similar\_fragment}(\mathbf{e}1, \mathbf{e}2)\right\| > \left\|\mathbf{similar\_fragment}(\mathbf{e}1, \mathbf{e}3)\right\| \end{array} \right.$$

Equation A 13. Copy of a code fragment

In terms of our methodology, $\mathbf{cloned\_by\_treshold}$ is a predicate that given a method $\mathbf{e}$ and a number $\mathbf{t}$ says if the method contains a similar fragment of a length equal or greater than $\mathbf{n}$ tokens with any other method in the application.

$$cloned\_by\_threshold : E^3 \times \aleph \rightarrow Boolean$$
$$cloned\_by\_threshold(e,t) = t \leq \left\| similar\_fragment(e, copy(e)) \right\|$$

Equation A 14. Clone relation between methods by length of their copies

The function ***clone_start*** returns the index of the first token in the given method ***e1*** where the cloned fragment starts.

$$clone\_start : E^3 \rightarrow \aleph$$
$$clone\_start(e1) = j \in \aleph \left| \begin{array}{l} \forall e2_i^1 \in similar\_fragment(copy(e1), e1) \\ e1_{j+i}^1 = e2_i^1 \end{array} \right.$$

Equation A 15. Clone start. First token cloned.

Our definition of being cloned requires the cloned fragment in the method to comply with several conditions in order to discard false positives. These conditions are stated in the formula of the predicate ***cloned***. The first condition (***cloned_by_treshold(e,30)***) describes precisely the type and level of similarity required among tokens of the method and the copy. The fragments shared should be a sequence of tokens of at least 30 tokens of the same type, producing the same code structure, i.e. with identical syntax tokens. This condition establishes the minimum similarity required. The other five conditions establish the maximum number of differences tolerated among fragments. The second condition (***¬generated(e)***) establishes that generated methods should not be considered clones, given that they do not require consistent updated. The third condition states that the percentage of syntax tokens in the cloned fragments should be less than 45% of the length of the fragment to avoid purely structural similarities (***syntax_tokens_%(e,copy(e))<0.45***). The fourth condition states that the percentage of literal tokens should be less than 45% of the length of the fragment to avoid clones just based on values of variables (***literal_tokens_%(e,copy(e))***). The fifth condition state that the percentage of tokens that differ between the cloned fragments should be less than 45% of the length of the fragment to avoid fragments that are similar by accident (***diff_tokens_%(e,copy(e))***). The sixth and final condition states that the percentage of tokens that refer to methods or types and that differ between the cloned fragments should be less than 45% of the tokens that refer to methods or types to avoid clones with different meaning (***diff_method-type_tokens_%(e,copy(e))***).

$$\mathbf{cloned} : \mathbf{E}^3 \rightarrow \mathbf{Boolean}$$

$$\mathbf{cloned}(\mathbf{e}) = \begin{pmatrix} \mathbf{cloned\_by\_threshold}(\mathbf{e}, 30) \\ \wedge \neg \mathbf{generated}(\mathbf{e}) \\ \wedge \mathbf{syntax\_\%}\begin{pmatrix}\mathbf{similar\_fragment}(\mathbf{e}, \mathbf{copy}(\mathbf{e})),\\ \mathbf{clone\_start}(\mathbf{e})\end{pmatrix} < 0.45 \\ \wedge \mathbf{literal\_\%}\begin{pmatrix}\mathbf{similar\_fragment}(\mathbf{e}, \mathbf{copy}(\mathbf{e})),\\ \mathbf{clone\_start}(\mathbf{e})\end{pmatrix} < 0.45 \\ \wedge \mathbf{diff\_\%}\begin{pmatrix}\mathbf{e},\\ \mathbf{similar\_fragment}(\mathbf{e}, \mathbf{copy}(\mathbf{e})),\\ \mathbf{clone\_start}(\mathbf{e})\end{pmatrix} < 0.45 \\ \wedge \mathbf{diff\_method\_type\_\%}\begin{pmatrix}\mathbf{e},\\ \mathbf{similar\_fragment}(\mathbf{e}, \mathbf{copy}(\mathbf{e})),\\ \mathbf{clone\_start}(\mathbf{e})\end{pmatrix} < 0.45 \end{pmatrix}$$

Equation A 16. Cloned

The percentage of syntax tokens (**syntax_%(cf)**) is defined as the number of syntax tokens in the cloned fragment over the number of tokens in the cloned fragment.

$$\mathbf{syntax\_\%} : \mathbf{seq}(\mathbf{E}^1) \rightarrow \Re$$

$$\mathbf{syntax\_\%}(\mathbf{cf}) = \frac{\sum_{\mathbf{i}=0}^{\mathbf{i} \le \|\mathbf{cf}\|} \left(\mathbf{e_i} \in \mathbf{cf} \wedge \mathbf{syntax}\left(\mathbf{e_i}\right)\right) \Rightarrow 1}{\|\mathbf{cf}\|}$$

Equation A 17. Percentage of syntax tokens in a cloned fragment

The percentage of literal tokens (**literal_%(cf)**) is defined as the number of literal tokens in the cloned fragment over the number of tokens in the cloned fragment.

$$\mathbf{literal\_\%} : \mathbf{seq}(\mathbf{E}^1) \rightarrow \Re$$

$$\mathbf{literal\_\%}(\mathbf{e1}, \mathbf{cf}) = \frac{\sum_{\mathbf{i}=0}^{\mathbf{i} \le \|\mathbf{cf}\|} \left(\mathbf{e_i} \in \mathbf{cf} \wedge \mathbf{literal}\left(\mathbf{e_i}\right)\right) \Rightarrow 1}{\|\mathbf{cf}\|}$$

Equation A 18. Percentage of literal tokens in a cloned fragment

The percentage of different tokens between the method **e1** and its clone **cf** (**diff_%(e1,cf)**) is defined as the number of tokens that are different among the cloned fragments, over the number of tokens of the cloned fragment.

$$\mathit{diff\_\%} : E^3 \times \mathit{seq}\left(E^1\right) \times \aleph \rightarrow \Re$$

$$\mathit{diff\_\%}(e1, cf, j) = \frac{\sum_{i=0}^{i \leq \|cf\|} \left(e1^1_{j+i} \neq cf_i\right) \Rightarrow 1}{\|cf\|}$$

Equation A 19. Percentage of different tokens among cloned fragments

The percentage of different method-type tokens between the method ***e1*** and its clone ***cf*** (***diff_method-type_%(e1,cf)***) is defined as the number of tokens that are different among the cloned fragments and refer to methods or types, over the number of tokens that refer to methods or types of the cloned fragment.

$$\mathit{diff\_method-type\_\%} : E^3 \times \mathit{seq}\left(E^1\right) \times \aleph \rightarrow \Re$$

$$\mathit{diff\_method-type\_\%}(e1, e2, j) = \frac{\mathit{diff\_method\_\%}(e1, e2, j) + \mathit{diff\_type\_\%}(e1, e2, j)}{2}$$

Equation A 20. Average percentage of different tokens among cloned fragments from the tokens representing types and method calls

The percentage of different method-tokens between the method ***e1*** and its clone ***cf*** (***diff_method _%(e1,cf)***) is defined as the number of method-tokens that are different between the method ***e1*** and its clone ***cf***, over the number of method-tokens in the cloned fragment.

$$\mathit{diff\_method\_\%} : E^3 \times \mathit{seq}\left(E^1\right) \times \aleph \rightarrow \Re$$

$$\mathit{diff\_method\_\%}(e1, cf, j) = \frac{\sum_i \begin{pmatrix} \mathit{method}\left(e1^1_{j+i}\right) \\ \wedge\ \mathit{method}\left(cf_i\right) \\ \wedge\ e1^1_{j+i} \neq cf_i \end{pmatrix} \Rightarrow \mathit{similarity}\left(e1^1_{j+i}, cf_i\right)}{\sum_i \begin{pmatrix} \mathit{method}\left(e1^1_{j+i}\right) \\ \wedge\ \mathit{method}\left(cf_i\right) \end{pmatrix} \Rightarrow 1}$$

Equation A 21. Percentage of different tokens among cloned fragments from the tokens representing method calls

The percentage of different type-tokens between the method ***e1*** and its clone ***cf*** (***diff_type _%(e1,cf)***) is defined as the number of type -tokens that are different between the method ***e1*** and its clone ***cf***, over the number of type -tokens in the cloned fragment.

$$diff\_type\_\%: E^3 \times seq\left(E^1\right) \times \aleph \rightarrow \Re$$

$$diff\_type\_\%(e1, cf, j) = \frac{\sum_i \begin{pmatrix} type\left(e1^1_{j+i}\right) \\ \wedge\ type\left(cf_i\right) \\ \wedge\ e1^1_{j+i} \neq cf_i \end{pmatrix} \Rightarrow similarity\left(e1^1_{j+i}, cf_i\right)}{\sum_i \begin{pmatrix} type\left(e1^1_{j+i}\right) \\ \wedge\ type\left(cf_i\right) \end{pmatrix} \Rightarrow 1}$$

Equation A 22. Average percentage of different tokens among cloned fragments from the tokens representing types

Finally, the similarity between two tokens ***e1*** and ***e2*** is defined as the number of consecutive characters that the tokens share, over the number of consecutive characters among the tokens.

$$similarity : E^1 \times E^1 \rightarrow \Re$$

$$similarity(e1, e2) = \frac{\sum_{i=0}^{i<\min(\|e1\|,\|e2\|)} \left(e1^0_i = e2^0_i \wedge e1^0_{i+1} = e1^0_{i+1}\right) \Rightarrow 1}{\left(\|e1\| * \|e2\|\right)}$$

Equation A 23. Similarity between two tokens

A.3 Evolution of SCIs

This section of the appendix presents the formulas relevant to explain the concepts behind the analysis of the evolution of the SCIUS, that is, the analysis over time of its extension, persistence and stability.

A.3.1 Extension

Extension per commit transaction: it is the average extension of the SCIUS in the application at that snapshot. The function ***extension_contribution*** gives the percentage of SCEs in the application that have the SCIUS at a snapshot ***i***, of the SCEs that compose the application at the snapshot ***i***.

$$extension\_application : \aleph \rightarrow \Re$$

$$extension\_application(i) = \frac{\left\|\left\{e \in s_i \mid hasSCIUS(e, i)\right\}\right\|}{\|s_i\|}$$

Contribution of a SCE of higher granularity than the one analyzed to the extension of SCIUS in

the application: it is the extension of the SCIUS in the application, by the given snapshot ***i***, due to the SCE analyzed (***e1***). The function ***extension_contribution*** gives the percentage that a SCE ***e*** contributes to the SCIUS in the application at the commit transaction ***i***.

$$\mathbf{extension\_contribution} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{extension\_contribution}(i, e1) = \frac{\left\| \{ e2 \in e1 \mid \mathbf{hasSCIUS}(e2, i) \} \right\|}{\left\| \{ e3 \in s_i \mid \mathbf{hasSCIUS}(e3, i) \} \right\|}$$

Extension in SCEs

Another possible definition for extension is the number of components of a SCE affected by the SCIUS, from all the components of the SCE. This definition is only valid in case the SCIUS is defined in terms of components of the SCE to be analyzed. For instance, if the SCE to be analyzed are classes, but the SCIUS occurs at the level of methods, e.g. long parameter list.

Extension by SCE: it is the average extension of the SCIUS in the SCEs of the application at the given commit transaction. The function ***extension_SCE*** gives the percentage of components of the SCE ***e*** that have the SCIUS at a commit transaction ***i***. That is, the number of components of the SCE ***e*** that have the SCIUS (***size_scius*** as defined below), over the number of components of the SCE ***e*** (***size_SCE***).

$$\mathbf{extension\_SCE} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{extension\_SCE}(i) = \frac{\sum_{e \in Si \wedge (\mathbf{hasSCIUS}(e,i)=1)} \frac{\mathbf{size\_scius}(e,i)}{\mathbf{size\_SCE}(e,i)}}{\sum_{e \in Si \wedge (\mathbf{hasSCIUS}(e,i)=1)} 1}$$

The function ***size_scius*** gives the number of SCEs of lower level that compose ***e*** at the snapshot ***i***, and that had the SCIUS.

$$\mathbf{size\_scius} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{size\_scius}(i, e) = \sum_{(e2 \in e) \wedge (\mathbf{scius}(e2)) \wedge (\mathbf{level}(e2) < \mathbf{level}(e))} 1$$

The function ***size_SCE*** gives the number of SCEs of lower level that compose ***e*** at the snapshot ***i***.

$$\mathit{size\_SCE} : \aleph \times E \rightarrow \Re$$

$$\mathit{size\_SCE}(i, e) = \sum_{(e2 \in e) \wedge (\mathit{level}(e2) < \mathit{level}(e))} 1$$

A.3.2 Persistence

Persistence is the percentage of the lifetime of a SCE that includes a SCIUS. It aims to show the longevity of the SCIUS in the application. The graphs proposed to analyze it are:

Persistence per commit transaction: average lifetime percentage of the SCEs with the SCIUS by the given commit transaction. The function ***persistence_application*** gives the average persistence of SCEs affected by the SCIUS at the given snapshot ***i***.

$$\mathit{persistence\_application} : \aleph \rightarrow \Re$$

$$\mathit{persistence\_application}(i) = \frac{\sum_{e \in \mathit{SCE\_affected\_by\_SCIUS}(i)} \mathit{persistence\_SCE}(e, i)}{\left\| \mathit{SCE\_affected\_by\_SCIUS}(i) \right\|}$$

The function ***SCE_affected_by_SCIUS*** gives a set of SCEs that have been affected by the SCIUS by the given snapshot ***i***.

$$\mathit{SCE\_affected\_by\_SCIUS} : \aleph \rightarrow \mathit{set}(E)$$

$$\mathit{SCE\_affected\_by\_SCIUS}(i) = \left\{ e \in E \;\middle|\; \left(\sum_{j=1}^{i} \begin{cases} \mathit{hasSCIUS}(e, j) \Rightarrow 1 \\ (\neg \mathit{hasSCIUS}(e, j)) \Rightarrow 0 \end{cases} \right) \geq 1 \right\}$$

The function ***persistence_SCE*** gives the percentage of snapshots of the lifetime of the SCE ***e***, that it has had the SCIUS by the given snapshot ***i***.

$$\mathit{persistence\_SCE} : E \times \aleph \rightarrow \Re$$

$$\mathit{persistence\_SCE}(e, j) = \frac{\mathit{snapshots\_with\_SCIUS}(e, j)}{\mathit{snapshots\_alive}(e, j)}$$

The function ***snapshots_with_SCIUS*** counts the number of snapshots that a SCE ***e*** have had the SCIUS by the given snapshot ***i***.

$$\mathit{snapshots\_with\_SCIUS} : E \times \aleph \rightarrow \aleph$$

$$\mathit{snapshots\_with\_SCIUS}(e, j) = \sum_{i=1}^{j \leq i} \mathit{hasSCIUS}(e, i)$$

The function ***commits_alive*** counts the number of snapshots that the SCE ***e*** has been part of the system by the given snapshot ***i***.

$$\mathbf{snapshots\_alive} : \mathbf{E} \times \aleph \rightarrow \aleph$$

$$\mathbf{snapshots\_alive}(\mathbf{e}, \mathbf{j}) = \sum_{\mathbf{i}=1}^{\mathbf{i} \leq \mathbf{j}} \begin{cases} \mathbf{e} \in \mathbf{Si} \Rightarrow 1 \\ \mathbf{e} \notin \mathbf{Si} \Rightarrow 0 \end{cases}$$

Persistence of a SCE of higher granularity level than the one analyzed: it is the average persistence of the SCEs that form the given SCE analyzed (***e1***) by the given snapshot ***i***. The function ***persistence_contribution*** gives the average persistence of the SCEs ***e2*** that compose the SCE ***e1*** by the snapshot ***i***. That is, the average persistence of the SCEs ***e2*** that compose the SCE ***e1*** (i.e. the average persistence of the ***e2*** ∈ ***e1***).

$$\mathbf{persistence\_contribution} : \aleph \times \mathbf{E} \rightarrow \Re$$

$$\mathbf{persistence\_contribution}(\mathbf{i}, \mathbf{e}1) = \frac{\sum_{\mathbf{e}2 \in \mathbf{e}1 \wedge \mathbf{e}2 \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} \mathbf{persistence\_SCE}(\mathbf{e}2, \mathbf{i})}{\sum_{\mathbf{e}2 \in \mathbf{e}1 \wedge \mathbf{e}2 \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} 1}$$

A.3.3 Stability

Stability is the percentage of changes to a SCE that occur inside the SCIUS, it aims to show the variability that the SCIUS introduces in the application. The graphs proposed to analyze stability are:

Stability by history point: it is the average stability of the SCEs in the application while having the SCIUS by the snapshot ***i***. The function ***stability_application*** gives the average stability of SCEs affected by the SCIUS at the given snapshot ***i***.

$$\mathbf{stability\_application} : \aleph \rightarrow \Re$$

$$\mathbf{stability\_application}(\mathbf{i}) = \frac{\sum_{\mathbf{e} \in \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i})} \mathbf{stability\_SCE}(\mathbf{e}, \mathbf{i})}{\left\| \mathbf{SCE\_affected\_by\_SCIUS}(\mathbf{i}) \right\|}$$

The function ***stability_SCE*** gives the percentage of changes to SCE ***e*** that have occurred in the components of e that have the SCIUS by the snapshot ***i*** . That is, the number of changes

of the SCE e that have occurred inside the SCIUS (***changes_by***), over the number of changes on the SCE e while it has the SCIUS (***changes_when_SCIUS***).

$$stability\_SCE : E \rightarrow \Re$$

$$stability\_SCE(i) = \sum_{e \in SCE\_affected\_by\_SCIUS(i)} \frac{\|changes\_when\_SCIUS(e,i)\|}{\|changes\_by(e,i)\|}$$

The function ***changes_by*** gives the set of commit transactions in which the SCE e is modified until the given commit transaction i.

$$changes\_by(e,i) : E \times \aleph \rightarrow set(\aleph)$$
$$changes\_by(e,i) = \{ j \in \aleph \mid (j > 0) \wedge (j \leq i) \wedge (e \in c_j) \}$$

The function ***changes_when_SCIUS*** gives the set of commit transactions in which the SCE e is modified, having the SCIUS, by the given commit transaction i.

$$changes\_when\_SCIUS(e,j) : E \times \aleph \rightarrow set(\aleph)$$
$$changes\_when\_SCIUS(e,j) = \{ i \in changes\_by(e,j) \mid hasSCIUS(e,i) \}$$

Contribution of SCE of higher granularity than the one analyzed to the stability of the application: it is the average stability of the SCEs that compose the given SCE $e1$ by the given snapshot i. The function ***stability_contribution*** gives the percentage that a SCE $e1$ contributes to the stability of the SCIUS in the application by the snapshot i. That is, the average stability of the SCEs $e2$ that compose the SCE $e1$ (i.e. the average stability of the $e2 \in e1$).

$$stability\_contribution : \aleph \times E \rightarrow \Re$$

$$stability\_contribution(i,e1) = \frac{\sum_{e2 \in e1 \wedge e2 \in SCE\_affected\_by\_SCIUS(i)} stability\_SCE(e2,i)}{\sum_{e2 \in e1 \wedge e2 \in SCE\_affected\_by\_SCIUS(i)} 1}$$

Stability in SCEs

Another possible definition for stability is the number of changes in the SCE inside the components of the SCE affected by the SCIUS, from all the changes of the SCE when it has the

SCIUS. This definition aims to assess the percentage of changes due to the SCIUS. This definition is only valid in case the SCIUS is defined in terms of components of the SCE to be analyzed.

Stability by SCE: it is the average stability of the SCE due to the SCIUS by the given snapshot ***i***. The function ***stability_SCE*** gives the percentage of changes to SCE ***e*** that have occurred in the components of ***e*** that have the SCIUS by the snapshot ***i***. That is, the number of changes of the SCE ***e*** that have occurred inside the SCIUS (***changes_in_SCIUS***), over the number of changes on the SCE ***e*** while it has the SCIUS (***changes_when_SCIUS***).

$$stability\_SCE : E \times \aleph \rightarrow \Re$$

$$stability\_SCE(e, i) = \frac{\|changes\_in\_SCIUS(e, i)\|}{\|changes\_when\_SCIUS(e, i)\|}$$

The function ***changes_in_SCIUS*** gives the set of commit transactions in which any components of the SCE ***e*** with the SCIUS have been modified, by the given commit transaction ***i***.

$$changes\_in\_SCIUS : E \times \aleph \rightarrow \aleph$$

$$changes\_in\_SCIUS(e, i) = \left\{ j \in changes\_when\_SCIUS(e, i) \;\middle|\; (e2 \in e) \wedge (e2 \in ctInSCIUS(j)) \right\}$$

The function ***ct_in_SCIUS*** gives the set of components of SCEs that have the SCIUS and that have been modified at a given commit transaction ***i***.

$$ct\_in\_SCIUS : \aleph \rightarrow set(E)$$

$$ct\_in\_SCIUS(i) = \left\{ e1 \in s_i \;\middle|\; (e2 \in e1) \wedge (hasSCIUS(e2, i)) \right\}$$

A.4 Changeability metrics

This section of the appendix presents the formulas relevant to explain the concepts related with the changeability metrics.

A **period** is a set of logical changes, i.e. ***P:set(C)***. The logical changes composing a period are not necessarily sequential, which permits to reason about SCEs that have the SCIUS intermittently.

Likelihood indicates whether a SCE changes more than other SCEs. The likelihood is defined

as the number of changes to the SCE ***e*** in the period ***p***, over, the number of changes in the period ***p***.

$$\mathit{likelihood} : E \times P \rightarrow \Re$$
$$\mathit{likelihood}(e, p) = \frac{\mathit{changes\_entity}(e, p)}{\mathit{changes\_period}(p)}$$

The number of changes a SCE ***e*** in a period ***p*** is defined as the number of times in which ***e*** changed in any of the commit transactions ***ct*** that composed the period ***p***.

$$\mathit{changes\_entity} : E \times P \rightarrow \aleph$$
$$\mathit{changes\_entity}(e, p) = \sum_{c \in p} \begin{cases} e \in c \Rightarrow 1 \\ e \notin c \Rightarrow 0 \end{cases}$$

The number of changes in the period ***p*** is defined as the number of SCEs of the granularity level analyzed that changed in commit transactions ***ct*** that composed the period ***p***.

$$\mathit{changes\_period} : P \rightarrow \aleph$$
$$\mathit{changes\_period}(p) = \sum_{c \in p} \|c\|$$

Frequency says how often the SCE changes. The frequency is defined as changes to the SCE ***e*** in the period ***p***, over, the number of logical changes in that period (***p***).

$$\mathit{frequency} : E \times P \rightarrow \Re$$
$$\mathit{frequency}(e, p) = \frac{\mathit{changes\_entity}(e, p)}{\|p\|}$$

Impact indicates the size of the ripple effect of changes that modify a SCE. Impact is the average number of SCEs that co-change with the SCE ***e*** in the period ***p***, over, the number of changes of ***e*** in the period ***p***. The average number of SCE that co-change with ***e*** is described below.

$$\mathit{impact} : E \times P \rightarrow \Re$$
$$\mathit{impact}(e, p) = \frac{\mathit{avg\_co\_changes}(e, p)}{\mathit{changes\_entity}(e, p)}$$

The average number of SCEs co-changes with the SCE ***e*** in the period ***p*** is defined as the number of co-changes with the SCE ***e*** ($\|c_i\|$) during the period ***p***, over, the number of SCEs in

the period p ($\|s_i\|$).

$$avg\_co\_changes : E \times P \rightarrow \Re$$

$$avg\_co\_changes(e, p) = \sum_{(e \in c_i) \wedge (c_i \in p)} \frac{\|c_i\|}{\|s_i\|}$$

Depth indicates to what extent the structure of the application hides changes. Depth is the percentage of SCEs that does not change with the other SCEs that compose a of higher granularity that contains all the SCE that changed on each logical change in the period p. The depth is the inverse of the average density of the changes of the SCE e in the period p. The average density indicates what percentage of the closest common ancestor of the SCEs changed in p was changed.

$$depth : E \times P \rightarrow \Re$$
$$depth(e, p) = 1 - avgDensity(e, p)$$

The average density of changes of a SCE e in the period p is defined as the sum of the density of the SCE e during p, over the number of times that the SCEs changed during the period p.

$$avgDensity : E \times P \rightarrow \Re$$

$$avgDensity(e, p) = \frac{\sum_{(e \in c) \wedge (c \in p)} density(e, c)}{changes\_entity(e, p)}$$

The density of changes of a SCE e in the period p is defined as the percentage of SCEs modified from the SCE that is the closest common ancestor of all the SCEs that were modified in a given logical change c.

$$density : E \times set(E) \rightarrow \Re$$

$$density(e, c) = \frac{\|c\|}{sces\_of\_the\_same\_level(e, closest\_common\_ancestor(c))}$$

The SCEs of the same level that the SCE analyzed $e1$ inside the common ancestor $e2$, is the number of SCEs $e3$ inside the common ancestor $e2$ that have the same level of granularity that the SCE analyzed $e1$.

$$sces\_of\_the\_same\_level : E \times E \to \aleph$$
$$sces\_of\_the\_same\_level(e1, e2) = \sum_{e3 \in e2 \wedge level(e1) = level(e3)} 1$$

The closest common ancestor of a set of SCEs modified in a logical change c, is the SCE of minimum level that contains all the SCEs in c. That is, is the element of common ancestors that given any other element in common ancestors, that other element has a higher level.

$$closest\_common\_ancestor : set(E) \to E$$
$$closest\_common\_ancestor(c) = e1 \left| \begin{array}{l} e1 \in common\_ancestors(c) \wedge \\ \left(\forall e2 \in common\_ancestors(c) \left| \begin{array}{l} e2 \neq e1 \\ \Rightarrow level(e2) > level(e1) \end{array} \right. \right) \end{array} \right.$$

The set of common ancestors of a set of entities is the intersection of the ancestors of each one of the entities.

$$common\_ancestors : set(E) \to set(E)$$
$$common\_ancestors(ct) = \bigcap_{e \in ct} ancestors(e)$$

The ancestors of a SCE e is the set of SCEs that have e as descendent.

$$ancestors : E \to set(E)$$
$$ancestors(e) = \begin{cases} \varnothing \ \vee \\ j : E \left| e \in j' \right. \end{cases}$$

Appendix B Statistical concepts used

This appendix summarizes statistical concepts used in this thesis.

B.1 Descriptive statistics

Descriptive statistics describe a group of numbers, by describing typical behavior of values in the group or by reducing the group to a few numbers that represent them.

B.1.1 Description of a sample

Any group of numbers can be described by the following set of numbers: minimum, first quartile, median, third quartile, and maximum. These five numbers indicate the range of the values in the group, their central tendency, and the spread/dispersion of the sample. The **minimum** is the lowest value in the group of numbers. The **maximum** is the highest value in the group of numbers. The **median** is the value that divides the numbers in the group in two equal parts, where all the numbers in one part are below the median, and all the numbers in the other part are above the median. To obtain the median is enough to order the values in the group, and then, take the value that is in the middle of the group. For instance, the median of 5,2,7,9,4 is 5. Note that the group ordered would be 2,4,5,7,9; and that the value in the middle is 5. However, if the amount of numbers in the group is even, the median is the average of the two values in the middle. For instance, if the group is 5,2,10,7,9,4 the median is 6. The group ordered would be 2,4,5,7,9,10. Note that 6 is the average between 5 and 7, which are the values in the middle of the ordered group. Finally, the **first and third quartiles** are respectively the 25^{th} and 75^{th} percentiles. **Percentiles** are an extension of the concept of median, which is also the 50^{th} percentile. Percentiles divide the group of numbers in 100 equal parts. The n-th percentile is the value for which n-% of the data is less than or equal to. For instance, the 25^{th} percentile is the value below which 25% of the numbers in the group lay. This also means that 75% of the data has a value above the value of the 25^{th} percentile. Therefore, **quartiles** indicate the numbers that divide the group of numbers into quarters. One quarter of the values is lower than the first quartile. One quarter of the values lay between the first quartile and the median. One quarter of the values is in between the median and the third quartile. Finally, one quarter of

the values is above the third quartile.

B.1.2 Box-plots

Box-plots depict the key values of a group of numbers to summarize the group graphically. Box-plots are plotted in parallel to compare features of distributions. Box-plots depict the key values of a distribution (see Figure A- 1) giving a visual indication of the range, spread and central tendency of the distribution. The meaning of the values shown by box-plots is explained above in section B.1.1. The box of the box-plot indicates that the values of 50% of the items are inside those boundaries (i.e. indicate the value of the first and third quartile). The black area inside the box shows the value in the median of the distribution. The inter-quartile range is the size of the box, and indicates the data spread of the values in the dataset. The values outside the inter-quartile range plus 50% are far from the rest of the values, and therefore are considered outliers. The larger the difference in the location of the box-plots, the more likely it is that the two item sets compared are different.

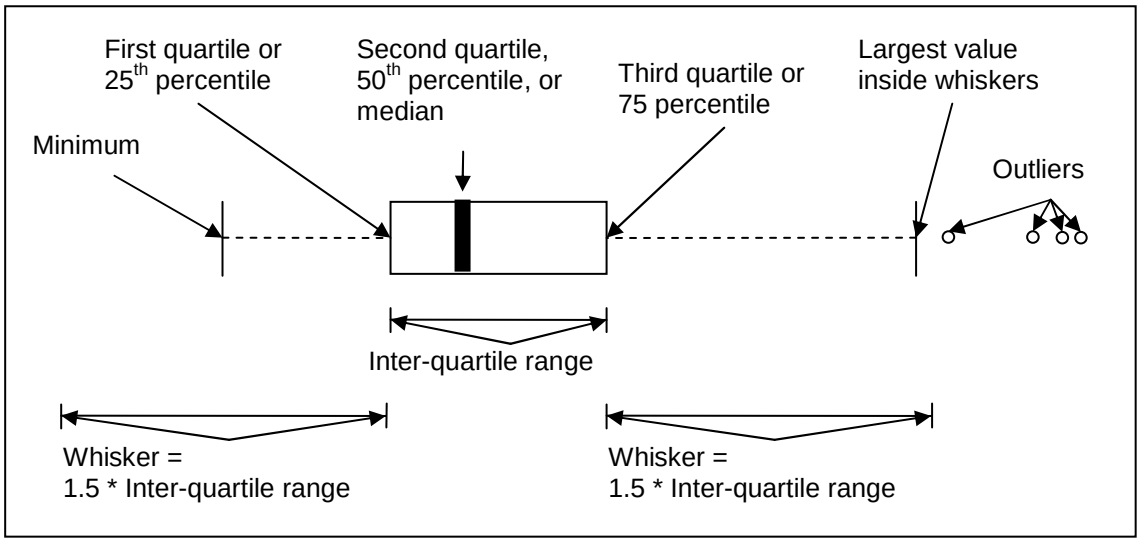


Figure A- 1. Box-plot parts.

B.1.3 Frequency distribution diagrams

Grouping numbers into frequency distributions illustrate the typical and atypical values in a group of numbers. A frequency distribution is an arrangement of the different values in a group of numbers ordered from the lowest to the highest; showing how frequent is each value in the group.

For example, suppose the following group of numbers:

2, 6, 9, 7, 5, 3, 1, 8, 5, 6, 4, 7, 5, 3, 8, 6, 4, 7, 2, 5, 6, 5, 4, 3, 6, 2, 8, 4, 7, 9, 1, 5, 3, 4

Once the group is clustered by values, and the clusters are ordered by value, the group of number is 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9.

Therefore, the frequency distribution says how frequent it is each value in the group:

Values:	1	2	3	4	5	6	7	8	9
Frequencies:	2	3	4	5	6	5	4	3	2

The frequency distribution graph for this example is shown in Figure A- 2. Frequency distribution graphs can be shown in two ways as a histogram, and as a density graph. Histograms depict a bar for each value or value interval, whose height represents the frequency. Density graphs depict a line that represents the probability of a value falling within the range. These two representations are analogous manners of showing which values are frequent/typical (in the case of the example 4, 5, and 6), and which values are unusual/atypical (in the example, 1, 2, 8 and 9).

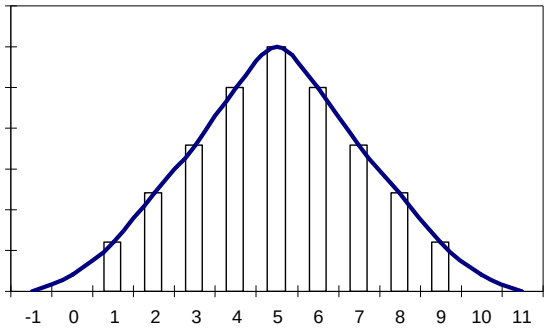


Figure A- 2. Frequency distribution graph. Depicted with bars is called a histogram, depicted with a line is called a density graph.

B.2 Inferential statistics

Inferential statistics aim to deduce the causes of groups of numbers. Inferential statistics are used to estimate about situations for which there is only partial information available. The

information available is called the sample; the complete information is called the population. Therefore, inferential statistics help to infer information about the population based on the characteristics of samples of that population.

B.2.1 Comparison of distributions using statistics

An example of inferential statistic is the set of tests that aim to establish if two data sets are significantly different. Significantly, in statistics, means that the probability of inference by chance is very low i.e. less than 5%. Usually the data sets compared are samples of the population under different circumstances, for instance, the population is the people suffering from depression; and the samples compared contain indicators of the depression symptoms on a set of people receiving a new drug, and a set of people receiving placebo. These tests can indicate two situations depending on the significance obtained. The first situation occurs when the test produces significant results. This means that the data sets compared are different. In terms of the example, it means that the new drug has a different effect on the depression symptoms than the placebo. The second situation occurs when the test does not produce significant results. This means that the results cannot be trusted. In other words, the statistical test cannot be certain of the differences between the data sets. The significance of a test is called the p-value. The p-value indicates the exact percentage of cases in which the similarity between distributions can be attributed to chance. As mentioned before, statisticians recommend a p-value below 0.05; however, in some cases 0.01 may also be valid.

To find if there is a difference between two data sets, it is necessary to choose the appropriate statistical test. The appropriateness of a statistical test depends on the properties of the distributions. There are two assertions to decide which test to apply. First, if the data sets are one value for the same subjects in the two circumstances analyzed the test should be *paired*; otherwise, the test should be *not-paired*. *Paired* tests are used when comparing the difference of two treatments in the same patient, *not-paired* tests are used when comparing the difference of two treatments in different patients, i.e. each patient undergoes only one treatment, and each data set has the patients using a particular treatment. Second, if the distributions are normal the test should be *parameterized*; otherwise the test should be not parameterized. Deciding if the test should be *parameterized* or not can be done by testing the normality (another statistical test) of the distributions to analyze. Table A 2 summarizes the types of tests according to the data-sets to compare, and their suitability according to the type of distributions obtained.

Table A 2. Classification of statistical tests according to their characteristics.

	Paired	Not paired
Parameterized	T-test paired	T- test not paired
Not parameterized	U- test/Wilcoxon-test	Mann Whitney- test

B.2.2 Clustering

CLARA is a clustering algorithm designed to process large datasets by processing the data by sub-sets of a fixed size. Each subset is divided into *k* clusters, and for each sub-set, the algorithm finds more representative objects of each cluster. Then, each object in the sub-set is assigned to the closest representative object. The sum of dissimilarities of the objects with respect to their representative object is calculated for each cluster to evaluate its quality. The representative objects chosen are those of the sub-set with the lowest sum of dissimilarities for all its clusters. The analysis of dissimilarities is repeated with the final partition.

B.2.3 Classification trees

A classification tree is partition algorithm that aims to predict an outcome from the characteristics of a dataset. In this way, the classification trees provide the set of common characteristics of the sets that it aims to predict. Classification trees divide recursively the dataset into two groups that are differentiated by a threshold in a characteristics of the dataset. The elements in the first group are above the threshold, and the elements in the second group are below the threshold. The characteristic and threshold chosen are those that separate better the elements depending on their value of the outcome that the classification tree is trying to predict. Therefore, if one wants to predict good grades for a set of students, the characteristic and threshold chosen each time would be the ones that produce a group composed mostly by students with high grades, and another group composed mostly by students with low grades. Each group obtained in afterwards divided again with the characteristic/threshold that divides best the groups in the characteristic to predict. Note that the optimal characteristic/threshold for a group, may be different from the optimal characteristic/threshold for its sibling group.

Appendix C Relation between method or clone characteristics and changeability metrics

This section presents the graphical and numeric tests to validate if the characteristics described in section 10.2.3.1 are correlated with changeability decay. The characteristics that presented the strongest relations with changeability decay are not presented in this appendix, but on section 10.2.3.2.

C.1 Method characteristics and changeability metrics

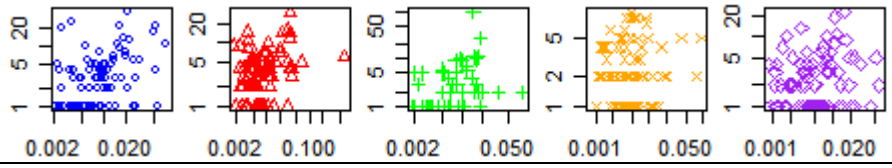
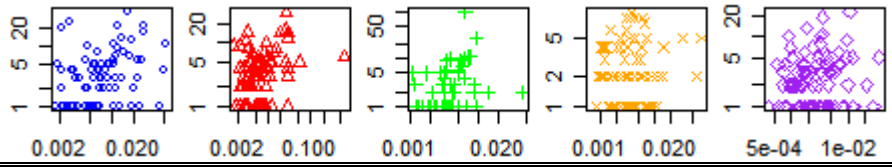
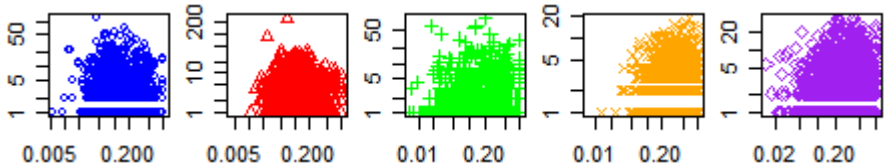
The characteristics related with likelihood, in order of strength of their relation, are: c*ommit created*, *loc*, *fan-out*, *fan-in*, and *nop*/*was cloned*. The characteristics related with frequency are: *commit created*, *loc*, *fan-out*, *complexity*, *nop*, and *fan-in*. Impact is inversely related to *loc*. Depth is inversely related with *loc*, *fan-in*, *fan-out*, and *complexity*.

The presentation of results is explained in Figure 10-15. Explanation of results table

.

C.1.1 Changeability metrics vs. complexity

Table A 3. Relation between changeability metrics and the complexity of the method

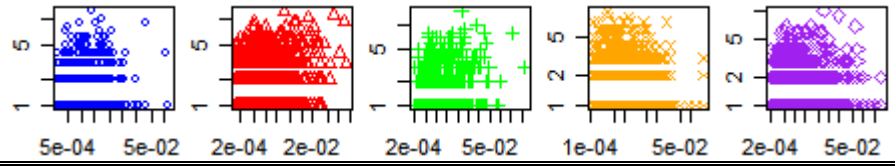
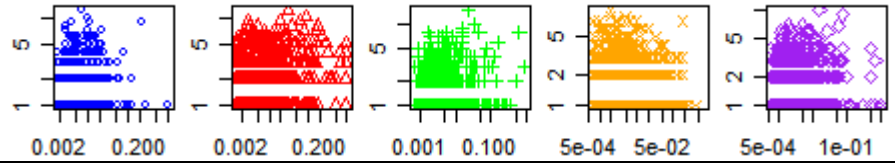
	Freecol	JEdit	Ganttproj.	Columba	JBoss
Frequency Cloned Direct 1.6					
Frequency SC Direct 1.6					
Depth All Inverse 1.2					

Complexity is directly related with the frequency of changes that cloned methods undergo (see Table A 3). Complexity is also inversely related with the depth of changes of any method. This means that, cloned methods that are complex tend to change more, co-changing with methods with in the same abstraction. In contrast, cloned methods that are simple change less but with methods far from them. However, the relation between complexity and changeability is weak.

C.1.2 Changeability metrics vs. number of parameters of the method

The number of parameters also seems to be directly related with the amount of changes of a method i.e. with likelihood and frequency (see Table A 4). Although, the relation could not be confirmed between cloned methods and likelihood, cloned methods do show a strong relation with the frequency of changes. This means that cloned methods have similar likelihood metrics regardless of the number of parameters, and that the higher the number of parameters, the more frequently would change cloned methods.

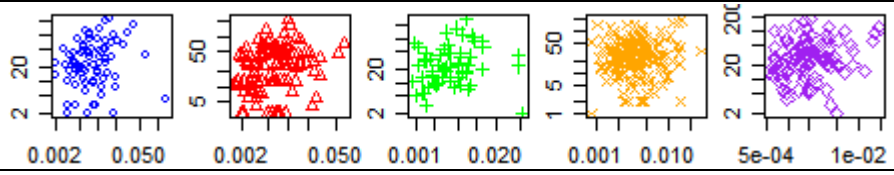
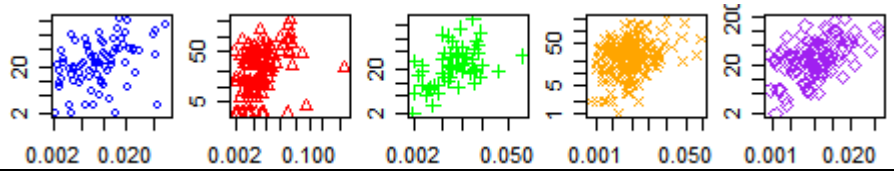
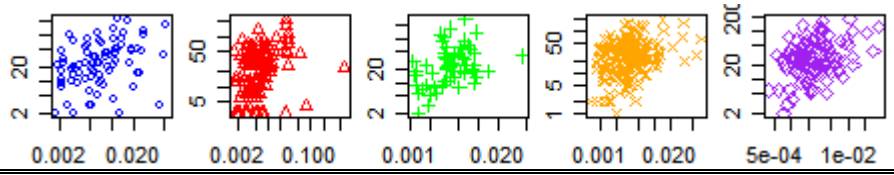
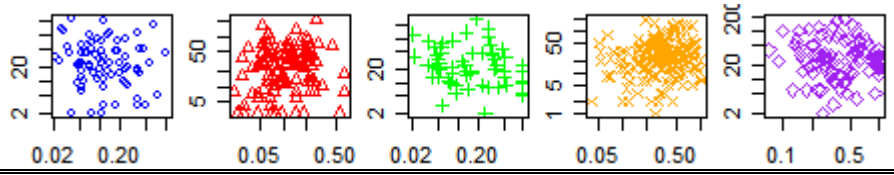
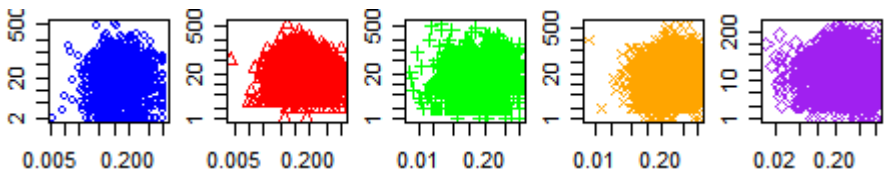
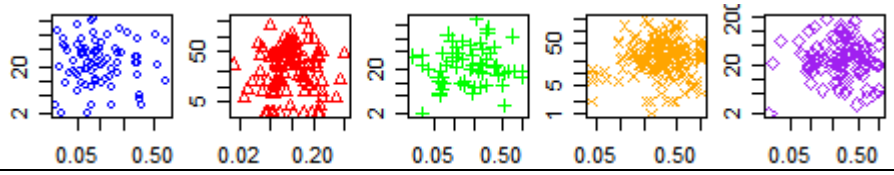
Table A 4. Relation between changeability metrics and the Number Of Parameters the method

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood All Direct 1.3					
Frequency All Direct 1.3					

C.1.3 Changeability metrics vs. size of the method

Large methods have a similar behavior than complex methods, that is, they change more but they change with closer methods (Table A 5). These results are also true for methods that are cloned. It is not clear if a lower complexity in changes (i.e. depth and impact) overcomes the disadvantages of the increase of changes. The strength of the LOC-Likelihood relation, and of the LOC-Frequency relation is higher that of LOC-Depth. Therefore, there is more chance of increasing the amount of changes than of decreasing the depth if the method is large. In consequence, having large methods is correlated with changeability decay.

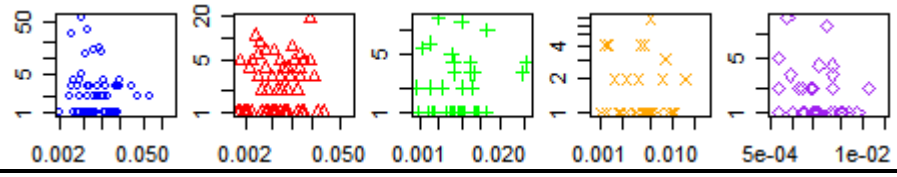
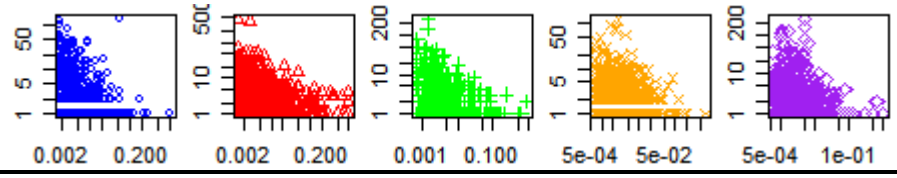
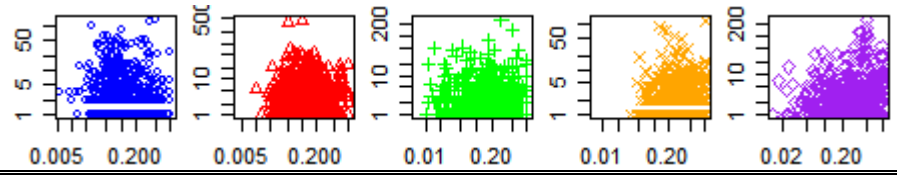
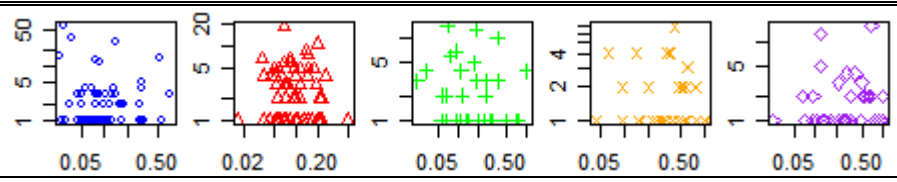
Table A 5. Relation between changeability metrics and the Lines Of Code of the method

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood SC Direct 1.4					
Frequency Cloned Direct 1.8					
Frequency SC Direct 1.5					
Impact SC Inverse 1.9					
Depth All Inverse 1.38					
Depth SC Inverse 1.5					

C.1.4 Changeability metrics vs. fan-in

The larger is the number of methods that call the method analyzed (*fan-in*), the lower is the frequency, and depth of the changes of the method analyzed (Table A 6). Inversely, methods with a low fan-in, would change more, more frequently, and with methods far away. If the method analyzed is cloned, having a large fan-in would decrease also its likelihood of change. This indicates that cloned methods, which are highly used, tend to change more than other cloned methods. In consequence, having a cloned fragment and having a high fan-in would increase the changeability decay of a method, because the method would require more changes to remain part of the application.

Table A 6. Relation between changeability metrics and the fan-in of the method

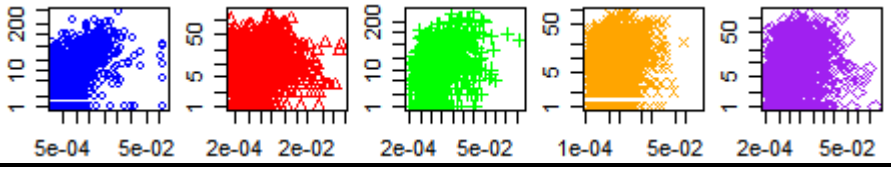
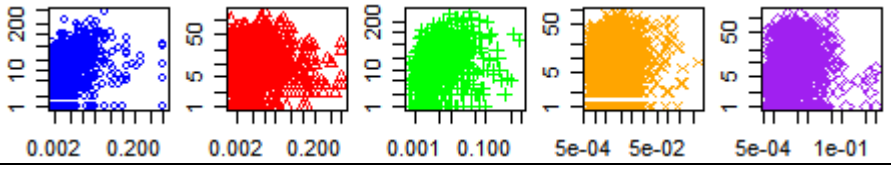
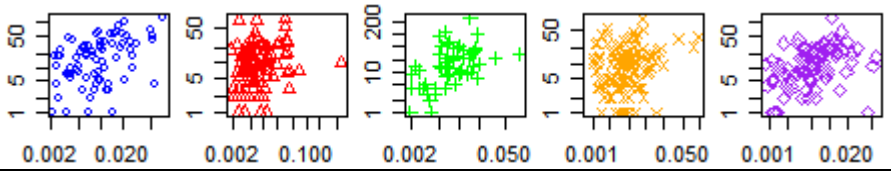
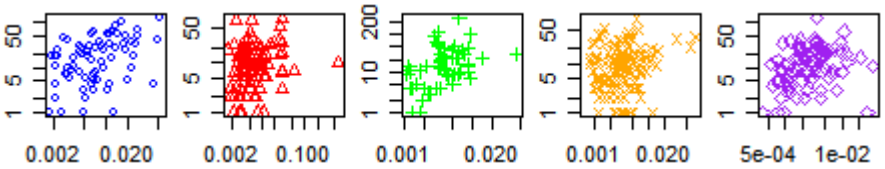
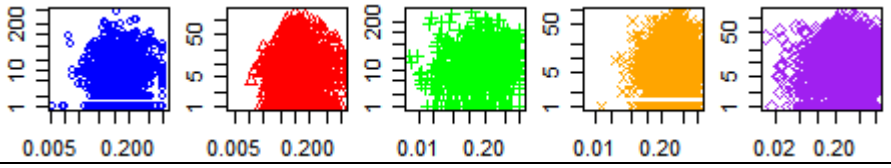
	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood SC Inverse 1.4					
Frequency All Inverse 1.2					
Depth All Inverse 1.1					
Depth SC Inverse 1.5					

C.1.5 Changeability metrics vs. fan-out

The larger is the number of methods that the analyzed method calls (*fan-out*), the higher is the likelihood, and frequency of change (see Table A 7). In other words, methods with a high fan-out change more and more frequently.

Cloned methods with high fan-out also change more frequently; but they change with closer methods than the cloned methods with an average fan-out.

Table A 7. Relation between changeability metrics and the fan-out of the method

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood All Direct 1.9					
Frequency All Direct 1.7					
Frequency Cloned Direct 1.7					
Frequency SC Direct 1.9					
Depth All Inverse 1.4					

C.2 Clone-related characteristics and changeability metrics

The characteristics related with likelihood, in order of strength of their relation, are: *%CCF*, *number of families*, *percentage affected*, and *percentage of method/type tokens that differ from the family*. The characteristics related with frequency are: *clone created*, *number of families*, *percentage affected*, *% of dissimilarity*, and *lifetime affected*. The clone-characteristics related with impact are: *percentage affected*, and *number of families* / *percentage of method/type tokens that differ from the family*. Depth is directly related with the *percentage affected*, the *size of the clone*, and the *commit cloned*.

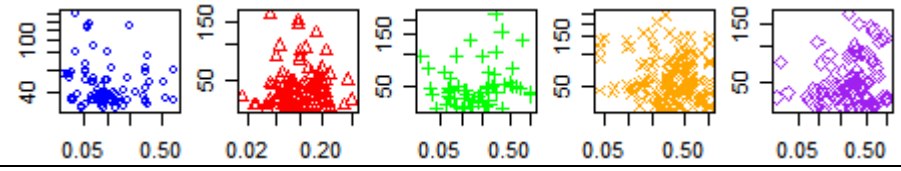
The presentation of results is explained in Figure 10-15. Explanation of results table

.

C.2.1 Changeability metrics vs. size of the clone

The size of the clone seems to be directly related with the distance to the methods that co-change with the method analyzed (see Table A 8). It is likely that this relation occurs because large clones are likely to be a result of copy-and-paste programming, and most of the methods that share a clone relation are close to each other (see section 8.2.9, and section 5.2.1.1). Again, this relation is weak, and therefore may not be useful to predict clones that increase the changeability of the methods that host them.

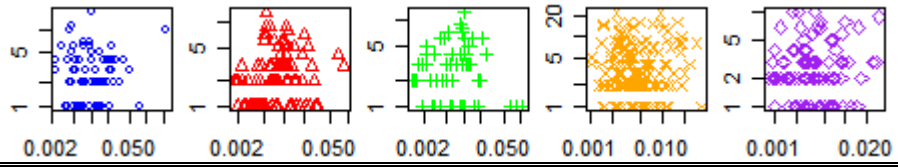
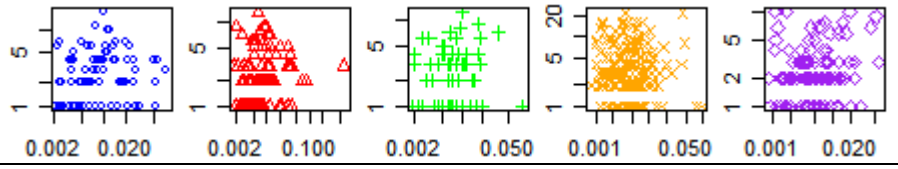
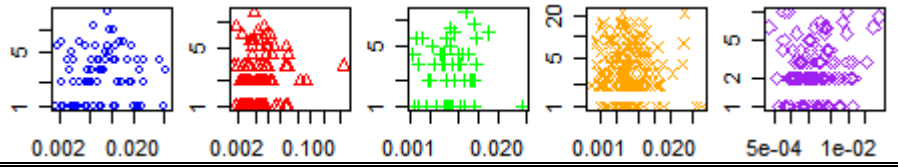
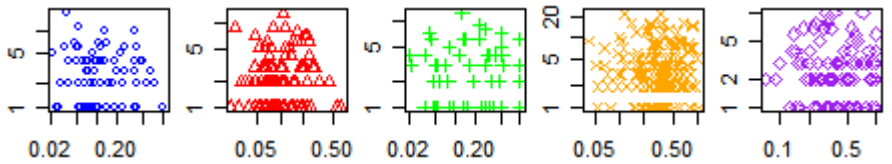
Table A 8. Relation between changeability metrics and the size of the clone

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Depth C Direct 1.2					

C.2.2 Changeability metrics vs. number of clone families to which the method belongs

Methods with fragments cloned of different families tend to have changes with a higher frequency, and likelihood (Table A 9). Having several clone relations means that the method must be checked more for consistent changes, i.e. if any of the fragments of any of the families with which it is cloned changes, it increases the chance for the method to require the same change. Furthermore, methods with cloned fragments of several families are usually related with overlapping families. This reason that could explain the decrease in the amount of methods that co-change, because the number of methods that share the clone relation is likely to be much lower than the number of cloned fragments related to the method.

Table A 9. Relation between changeability metrics and number of families in the method

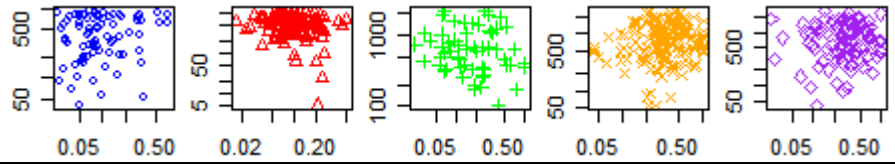
	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood C Direct 1.7					
Frequency C Direct 1.6					
Frequency SC Direct 1.7					
Impact SC Inverse 1.3					

C.2.3 Changeability metrics vs. commit in which the method becomes cloned

Methods cloned early in the lifetime of the application are more likely to be changed than methods cloned later in the lifetime of the application (Table A 10). This could be explained because the earlier they are cloned, the more changes of hidden relations they have to check to verify if their version of the clone should be also changed. Furthermore, as shown with the commit in which the method is created, at the beginning of the application's history changes are more frequent, probably because developers are not certain on how to implement the required functionality.

Note also that SC-methods have a weak, inverse relation with depth. This means that SC-methods cloned early in the application's lifetime tend to change with distant methods. Probably because introducing a clone with a distant method, late in the application's lifetime, is unlikely due to the size and complexity of the application.

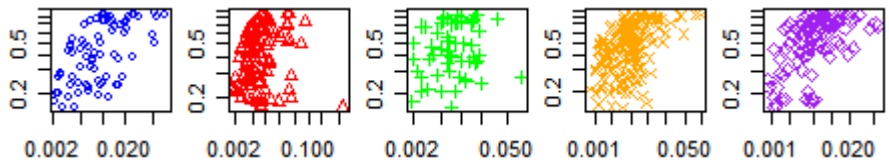
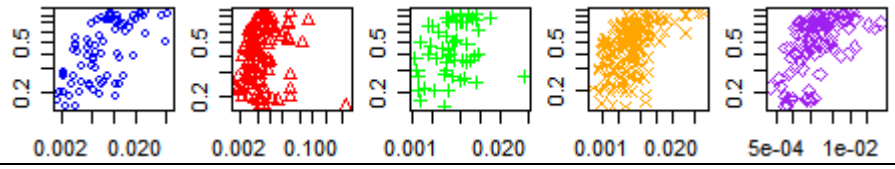
Table A 10. Relation between changeability metrics and the commit in which the method becomes cloned

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Depth SC Direct 1.1					

C.2.4 Changeability metrics vs. percentage of the method's lifetime in which it is cloned

The percentage of lifetime cloned is directly related to its frequency of changes (see Table A 11). Thus, methods that remain cloned for a large percentage of their lifetime tend to change more frequently. Therefore, while the method keeps the relations (in this case, clone relations) its need for changes increases.

Table A 11.Relation between changeability metrics and the percentage of the method's lifetime in which the method is cloned

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Frequency Cloned Direct 1.0					
Frequency SC Direct 1.06					

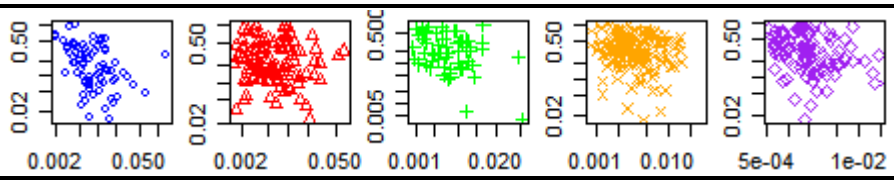
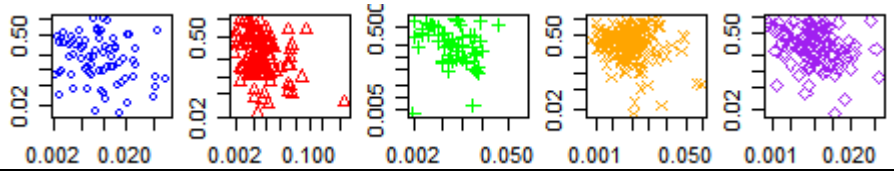
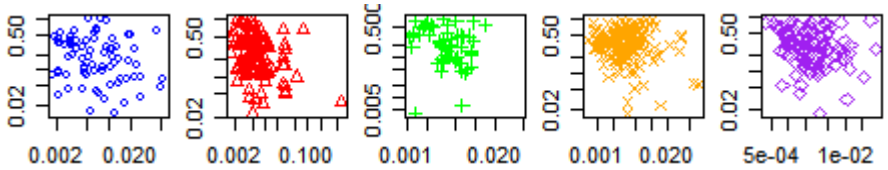
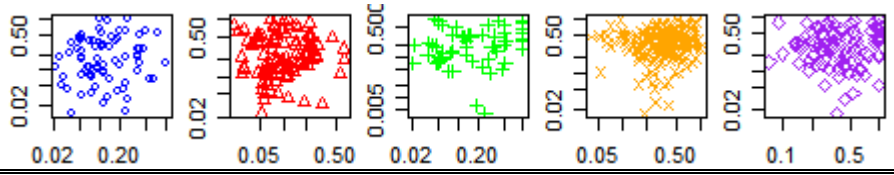
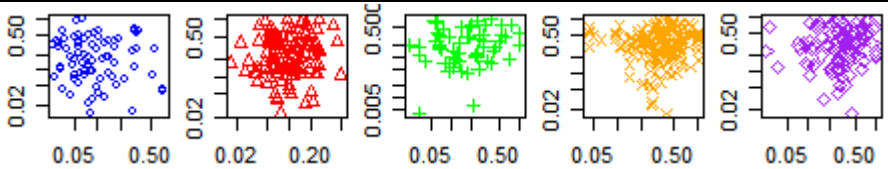
C.2.5 Changeability metrics vs. percentage of tokens cloned in the method

The coverage of the clone in the method is inversely related to the chance of the clone to be changed, but directly related to the impact and depth of its changes (Table A 12). This means that methods with low clone coverage change more, change more frequently, but tend to change alone or with a few, close methods. A possible reason for these relations is that methods that are partially cloned have more customizations in their cloned fragments than methods with a higher cloned coverage. This means that less of the changes in methods with fragments of the same family would affect it, but also that it changes might be more complex. Another possible explanation is that clones that cover a small percentage of the method are may be more likely to be forgotten for consistent changes with its family, and therefore are changed in isolation in late

propagations.

The fact that the relations between the percentage of tokens cloned in the method and the changeability metrics are found in SC-methods, indicates that the percentage of tokens cloned is a key characteristic in determining the harmfulness of a clone.

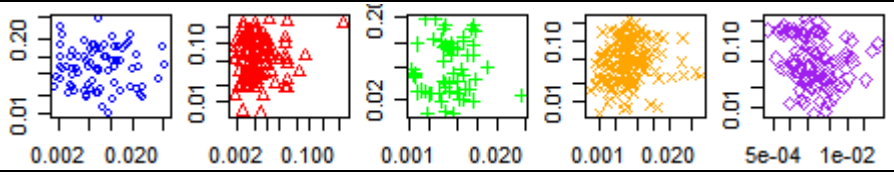
Table A 12. Relation between changeability metrics and the percentage of tokens cloned in the method

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood SC Inverse 1.4					
Frequency Cloned Inverse 1.3					
Frequency SC Inverse 1.5					
Impact SC Direct 1.9					
Depth SC Direct 1.7					

C.2.6 Changeability metrics vs. percentage of different tokens in the clone

The percentage of tokens that differ in cloned families seems to be directly related with the frequency of changes of the method (Table A 13). A possible explanation for this relation is that the lower the percentage of tokens that differ, the most likely it is for the cloned fragment to require a consistent change. Therefore, if there is a change in any of the cloned fragments with which the method is cloned, it is likely that the method would also require that change.

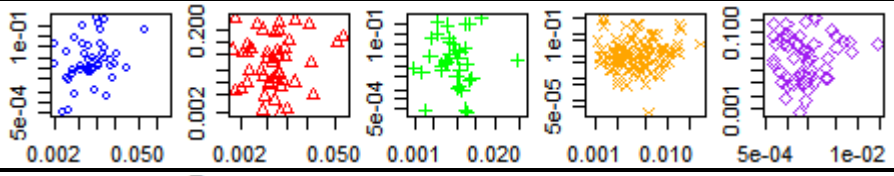
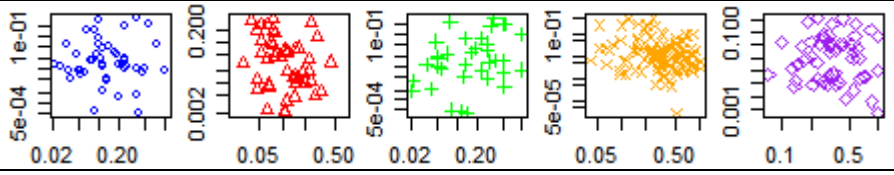
Table A 13. Relation between changeability metrics and the percentage of different tokens in the families of the method

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Frequency SC Direct1.5					

C.2.7 Changeability metrics vs. percentage of different method/type tokens in the clone

The percentage of differences in the methods and types referred in a cloned fragment is directly related with its likelihood of changes, and inversely related with the impact of its changes (Table A 14). Therefore, having a low percentage of differences in the methods and types referred in the cloned fragment tends to be more harmful for SC-methods, because they generate changes of a greater impact. Cloned fragments with a high percentage of differences in the methods or types called may indicate unavoidable clones due to insufficient modularity abstractions in the language. If this is the case, the clones are not accidental and require consistent changes provoking an increase in the number of methods that co-change with the cloned method. The fact that the relation is only present for SC-methods may indicate that clones with high percentage of different methods/types are not necessarily among the methods with higher impact. Therefore, the effect of a high percentage of different method/types in the cloned fragment with respect to other clones in its family does not have a strong relation with an increase of impact.

Table A 14. Relation between changeability metrics and the differences in methods and types in the family

	Freecol	JEdit	Ganttproj.	Columba	JBoss
Likelihood SC Direct 1.3					
Impact SC Inverse 1.3					

Appendix D Glossary

Adaptive maintenance: is a type of maintenance task to keep a computer program usable in a changed or changing environment.

Add: is a type of change in the history of a clone family, in which cloned fragments are added to the previous version of the clone family.

Age: is a characteristic of the relation between method and its clones. The age is defined as the commit transaction in which the first cloned fragment is added to the method.

Always with a source code Issue: is a subset of the SCEs (Source Code Entities) that have the SCIUS (Source Code Issue Under Study) all their lifetime. The set is identified by the acronym AI. In the case of clones, it is identified with the acronym AC that stands for methods Always Cloned.

Analysability: is a characteristic required to achieve maintainability. It is defined as the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

Anti-patterns: are solutions inappropriate for solving a problem. For example, inappropriate implementation patterns.

Box-plots: are graphs that depict the key values of a group of numbers to summarize the group graphically.

Branch: is a sequence of changes not merged to the main history line (or *trunk*) of a SCM application. Having branches implies that a file can have several versions at a moment in time.

Changeability: is a characteristic required to achieve maintainability. It is defined as the capability of the software product to enable a specified modification to be implemented.

Changeability decay: occurs when the effort or difficulty required to introduce a change increases. Other authors also consider that there is changeability decay whenever the quality of the code decreases.

Changes in a Family due to the Clone in a method (%CCF): is the ratio between the number of changes inside the cloned fragments of a method, and the number of changes of all the fragments in the families corresponding to those cloned fragments.

Changes in a Method due to its Clones (%CCM): is the ratio between the number of changes inside the cloned fragments of a method, over the number of changes of that method.

Characteristic: is a property of an entity at a moment in time. A particular entity has several attributes at a given moment in time. The attributes of a SCE (Source Code Entity) may include its name, size, if it has a SCI or not, if it was modified or not, etc.

Clone family: is the set of code fragments that share the same clone. It is also known as **clone class** or **clone cluster**.

Clone: is the longest sequential segment of code shared by at least two sections of code.

Clone relation: is a conceptual link between fragments of code. A clone relation occurs when the fragments are *similar*.

Cloned fragment (or clone instance): is a fragment of code that is *similar* to another fragment of code.

Close-down: is the maintenance stage when the system is not used anymore.

Commit created: is a characteristic of the method, defined as the commit transaction in which the method is created.

Commit transaction: is a set of changes done together (i.e. at the same time, by the same developer, with a defined purpose). It is also known as **logical change**.

Complexity or cyclomatic complexity (CC): is a characteristic of the method, defined as the number of branches inside the method.

Consistent change: is a type of change in the history of a clone family, in which all the fragments that compose the clone family change in the same way, at the same commit transaction.

Copy role: occurs when the method changes at the same commit transaction in which its first cloned fragment is added.

Corrective maintenance: is a type of maintenance task to corrects discovered faults.

Creation: is a characteristic of the relation between method and its clones. The creation is defined as the commit transaction in which the first clone relation is added to a method.

Creator: is a characteristic of the relation between method and its clones. The creator is

defined as the developer that created the first clone relation of a method.

CVS (Concurrent Versioning System): is a SCM application that does not store logical changes.

Dependencies fingerprints are a type of code representation used for clone detection. Dependencies fingerprints are useful for identifying clones in which the order of the statements may have been changed without changing their semantics, like in plagiarism analysis.

Depth indicates to what extent the structure of the application hides changes. Depth is a metric created to assess to what extent the commits in which a method changes are hidden behind an abstraction. The metric is called depth because if all changes are concentrated in a SCE of low level, for instance, in a class, it means that the change was made deep in the application, and that the change was oblivious to higher SCEs. In this way, the deeper a change is, the easier it is to implement. Depth is the percentage of SCEs that does not change with the other SCEs that compose a of higher granularity that contains all the SCE that changed on each logical change in a period. The depth is the inverse of the average density of the changes of the SCE e in the period .

Design flaws are source code characteristics that may affect maintainability factors. Design flaws include the composition of several bad smells, the violation of some Object Oriented principles and the lack of use of certain design patterns (like bridge, strategy, singleton, façade, etc.).

Dissimilarity: is a clone characteristic. It is defined as the ratio between the number of tokens that are different (between the cloned fragment and the clone that identifies the family), and the number of tokens of the cloned fragment. The dissimilarity of a method is the average dissimilarity of its cloned fragments along their lifetime.

Divergent change: is a type of change in the history of a clone family, in which only some of the fragments that compose the clone family change in the same way, at the same commit transaction.

Divergent Changes in a Family (%CDF): is the ratio between the number of *divergent changes* inside the clone-families related to the method, and the number of changes of the family corresponding to that method.

Divergent Changes in a Method (%CDM): is the ratio between the number of divergent

changes inside the clone-families related to the method, and the number of changes of that method.

Elimination: is a characteristic of the relation between method and its clones. The elimination is defined as the commit transaction in which the last cloned fragment of a method is eliminated, either because the fragment is deleted from the method, or because it is changed so that it is not similar to any other code fragment anymore.

Eliminator: is a characteristic of the relation between method and its clones. The elimination is defined as the developer that modifies the method in the commit transaction in which the last clone relation of the method is eliminated (see elimination).

Evolution: is the maintenance stage in software lifecycle in which the system is subject to fault correction and to adaptation to new user requirements or to a new environment. Evolution is a term also used for describing the dynamics of software history.

Extension of the SCIUS in the application: is the ratio between the number of SCEs that have the SCIUS at a given commit transaction and the number of SCEs that compose the application at that commit transaction.

Contribution to the extension of the SCIUS in a SCE: is the ratio between the number of SCEs inside the SCE analyzed that have the SCIUS at a given commit transaction, and the number of SCEs inside the SCE analyzed.

Extension of the SCIUS in the SCEs: is the average ratio between the number components of the SCE analyzed that have the SCIUS at a given commit transaction, and the number components of the SCE analyzed.

False negative: when the test to detect phenomena falsely indicates that the phenomena does not occur.

False positive: when the test to detect phenomena falsely indicates that the phenomena occurs.

Family size: is a clone characteristic that describes the number of fragments of code that share the same clone. In this document is approximated by the number of methods that contains clone instances of the same clone, regardless of the number of instances inside each method.

Fan-in: is a method characteristic that indicates the number of methods that call the method

analyzed.

Fan-out: is a method characteristic that indicates the number of methods that the method analyzed calls.

Frequency: is a measure of how often a SCE changes. The frequency is defined as changes to the SCE, over, the number of changes to the application (i.e. commit transactions) in a given period.

HasSCIUS: is a characteristic of the SCE that says if the SCE has the SCIUS or not at a moment in time.

History lines: are sequences of changes of an application stored in a SCM. An application can have several history lines at any moment in time. The main history line is called the trunk, and the history lines that are not the trunk are called branches.

History of an application: is sequence of snapshots ordered chronologically by commit transaction.

Impact: is a measure of the ripple effect of changes that modify a SCE. Impact is the average number of SCEs that co-change with the SCE, over, the number of changes to the application (i.e. commit transactions) in a given period.

Inconsistent change: it is a type of change in the history of a clone family, in which the only difference with the previous version of the clone family is that some clone instances do not change in the same way.

Independent evolution: it is a type of change in the history of a clone family, in which the only difference with the previous version of the clone family is that some clone instances change in a different way than the rest of the instances in the family. In many cases, an independent evolution can be identified as an independent change. The only way to confirm that there is an independent evolution is by ensuring that the subset of clone instances that change differently never become part of the family again.

Index: is an integer that indicates the chronological order of a commit transaction, that is, the history in which a logical change occurs.

Late propagation: it is a type of change in the history of a clone family, in which the only difference with the previous version of the clone family is that a subset of clone instances that

had a previous inconsistent change are modified in such a way that they become part of the family again.

Level: is the granularity of a SCE. Levels can be tokens, lines of code, methods, classes, files, modules, etc.

Literal percentage (or %Literals): it is a clone characteristic that describes the percentage of literals that have the clones inside a method. It is calculated as the ratio between the tokens that are literals in the cloned fragments of a method, and the tokens that compose the cloned fragments.

Lifetime affected (or %lifetime cloned): it is a clone characteristic that describes the percentage of commit transactions in which the method was cloned. It is calculated as the number of commit transactions of the method's lifetime in which the method has cloned fragments, over, the number of commit transactions in which the method is part of the application.

Likelihood: is a measure of the extent in which a SCE changes in comparison with other SCEs. The likelihood is defined as the number of changes to the SCE, over, the number of changes on any SCE of the application in a given period.

Lines Of Code (or LOC): it is a method characteristic that indicates the size of the method analyzed. It is calculated as the number of physical lines from the declaration to the end of the method.

Logical change: is a synonym for commit transactions. A logical change occurs when a developer sends the changes that she has made locally to the repository that stores the application (the SCM). In other words, a logical change can be identified in the SCM as a set of physical changes done with the same purpose, at the same time by a single developer. In the formalization of the metrics proposed, a logical change contains the set of entities that changed from one snapshot to the next snapshot in the history of the application.

Maintainability compliance: is the capability of the software product to adhere to standards or conventions related to maintainability. It is considered one of the necessary characteristics to consider an application maintainable.

Maintainability index: is a metric based on static analysis of the source code used as indicator of maintenance costs. The maintainability index is defined as a polynomial formula of

the number of operands and operators, the number of branches, the number of lines of code, and optionally the percentage of comments, all of them at module level.

Maintainability: is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. Maintainability is achieved through analyzability, changeability, stability, testability, and maintainability compliance.

Maintenance: is any activity after the first delivery of the application. Maintenance can be divided into several stages: evolution, servicing, phase-out, and close-down.

Meaningless clones: are duplicated code whose common fragments are incomplete blocks of code that are not semantically related. For instance, a method return followed by a method declaration, or the end of a conditional followed by the beginning of a loop.

Median: is the value that divides the numbers in the group in two equal parts, where all the numbers in one part are below the median, and all the numbers in the other part are above the median.

Method cloned: is a method that shares a fragment with at least another method in the application. That is, a method that contains at least one clone instance.

Method-Type percentage of differences (or mdiff): is a clone characteristic that refers to the percentage of method-call and type tokens that are different to the clone that identifies the family. It is calculated as the ratio between the method-call and type tokens in the clone instance that are different to the clone that identifies the family, over, the method-call and type tokens that compose the clone that identifies the family. If a method has several clone fragments, the value is the average of all clone fragments.

Methodology: is a systematic manner to accomplish tasks using a set of techniques and a set of rules to apply such techniques. A methodology should guide the definition of objectives, the organization of tasks, and decision of techniques to achieve the objectives fixed for the tasks [Nance '88].

Metrics fingerprints: is a type of source code representation used to detect clones. Metrics fingerprints are useful for lightweight and simple comparisons. However, the performance does not necessarily improve because the calculation of the metric may be very complex.

Never with a source code Issue (or NI): is a SCE never has the SCIUS. For our case

study, the methods never cloned are identified as NC-methods.

Number of families: is a characteristic to describe the relation between a method and its clones. It is calculated as the number of clone families with which the method is related (by its cloned fragments).

Number Of Parameters (or NOP): is a method characteristic that indicates the number of parameters received by the method analyzed.

Origin analysis: is a process to identify source code entities over time regardless of changes in their name or location.

Ownership: is a characteristic of a piece of code that refers to the developer that changes the most that piece of code in an interval of time. The ownership of the method refers to the developer that changes the method the most along its lifetime. The ownership of the clone refers to the developer that changes the method while it is cloned.

Percentage affected (or %tokens cloned or percentage cloned): is a characteristic to describe the relation between a method and its clones. It is calculated as the number of tokens cloned, over, the number tokens of the method.

Percentiles: are numerical values that divide a group of numbers in 100 equal parts. The n-th percentile is the value for which n-% of the data is less than or equal to.

Perfective maintenance: is a maintenance task that aims to improve the performance, maintainability, or other attributes of a computer program.

Persistence application: is a measure of the average persistence of SCEs affected by the SCIUS at the given commit transaction. That is, it is the ratio between the persistence of the SCIUS in the SCEs that have had the SCIUS up-to that commit transaction, over, the number of SCEs that have had the SCIUS up-to that commit transaction.

Persistence contribution: is a measure of the contribution of a SCE of high level of abstraction to the persistence of an application. It is calculated as the average persistence of the SCIUS on the SCEs that compose another SCE of a higher level of abstraction. That is, it is the ratio between the persistence of the SCIUS in the SCEs that compose to the SCE analyzed and that have had the SCIUS up-to that commit transaction, over, the number of SCEs that compose to the SCE analyzed and that have had the SCIUS up-to that commit transaction.

Phase-out: is the maintenance stage in which the application remains in use but no changes are performed.

Preventive maintenance: is a maintenance task that aims to detect and correct latent faults before they become faults.

Quartiles: are values that divide a group of numbers into quarters.

Relative risk (RR): is a metric used in epidemiological studies to assess if being exposed to an agent increases the risk of developing a disease [Green '00], for instance to check if smoking (agent) increases the chances of developing lung cancer (disease).

Role: is a characteristic to describe the relation between a method and its clones. It refers to the way in which the first cloned fragment of a method is created. There are three possible roles: seed, copy, and twins.

Same: is a type of change in the history of a clone family, in which there is no difference with the previous version of the clone family. That is, none of the fragments change, neither additions of fragments to the clone family nor deletions of fragments from the clone family.

Seed role: occurs when the first cloned fragment that a method has is created because another method copied it. It is detected by analyzing the changes in the methods that become part of the clone family when the clone is created. The cloned fragment analyzed is the seed of the clone; if it was not changed when it became cloned, and the other methods that acquire the clone at the same moment changed.

Semantic tokens: are those that represent instances of SCEs; such as, literals, types, and method names.

Servicing: is the maintenance stage in which only minor changes are possible to introduce in the application.

Shift: is a type of change in the history of a clone family, in which the only difference with the previous version of the clone family is that some cloned fragments change without becoming part of another clone family. That is, when they still share the same clone.

Similarity among code fragments: two fragments are similar if syntax tokens are identical, and if there is a correspondence in their semantic tokens.

Size: is a characteristic to describe the relation between a method and its clones. It refers to the

size of the clones of a method. It is calculated as the number of tokens cloned inside a method.

Sliding time window: is a time threshold used to identify commit transactions. A sliding time window of *X* minutes indicates that the maximum distance between two consecutive physical changes (done with the same rationale, by the same developer) it at most *X* minutes. Notice that this indicates that the length of all commit transactions is variable that is the reason for calling it a 'sliding window' in opposition to the 'fixed time window' in which the duration of a commit transaction has a fixed upper limit.

Snapshot: is the status of the application after a commit transaction. In the formalization of the metrics proposed, set of entities that compose the application after the moment in time defined by the commit transaction.

Sometimes with a source code Issue (or SI): is a SCE has had the SCIUS for a fraction of its lifetime. For our case study, the methods sometimes cloned are identified as SC-methods.

Source Code Entities (SCE): are the abstraction units of an application. Each SCE is composed of one or several SCEs of the same or of lower granularity. For example, in Java, packages are composed of other packages or classes, classes being of a lower level of granularity than packages.

Source Code Issue: is a source code characteristic considered as a harmful implementation of a Source Code Entity.

Stability: is the capability of the software product to avoid unexpected effects from modifications of the software. It is considered one of the necessary characteristics to consider an application maintainable.

Stability application: is a measure of the average stability of SCEs affected by the SCIUS at the given commit transaction. That is, it is the ratio between the stability of the SCEs that have had the SCIUS up-to that commit transaction, over, the number of SCEs that have had the SCIUS up-to that commit transaction.

Stability_contribution: is a measure of the contribution of a SCE of high level of abstraction to the stability of an application. It is calculated as the average stability of the SCEs that compose another SCE of a higher level of abstraction. That is, it is the ratio between the stability of the SCEs that compose to the SCE analyzed and that have had the SCIUS up-to that commit transaction, over, the number of SCEs that compose to the SCE analyzed and that have

had the SCIUS up-to that commit transaction.

Stability_SCE: is a measure of the instability on a SCE that can be attributed to its SCIUS. It is calculated as the average ratio between the number of changes in the components of the SCE that have the SCIUS up-to that commit transaction, over, the number of changes in the SCE up-to that commit transaction.

Subtract: is a type of change in the history of a clone family, in which the only difference with the previous version of the clone family is that at least one clone fragment is removed from the clone family. That is, the cloned fragment is deleted or when it changes enough so it cannot be considered anymore as part of the family because it does not share the clone that distinguishes the family.

Syntax fingerprints: is a type of source code representation used to detect clones. Syntax fingerprints exploit the syntax constraints of the programming language so that the comparison algorithms are aware of syntactic units allowing for a high precision.

Syntax percentage (%Syntax): is a clone characteristic that refers to the percentage of syntax tokens that compose the clone that identifies the family. It is calculated as the ratio between the number of syntax tokens, over, the number of tokens in the clone that identifies the family.

Syntax tokens: are those that cannot be instantiated, that is, keywords and operators.

System Configuration Management (SCM): is an application stores several versions of the same file. The SCM applications automatically save and merge the deltas between the versions uploaded by the developers and the version in the server.

Testability: is the capability of the software product to enable modified software to be validated. It is considered one of the necessary characteristics to consider an application maintainable.

Text fingerprints: is a type of source code representation used to detect clones. Text fingerprints are considered lightweight because they do not require validations based on the syntax of the programming language.

Tokens: is a type of source code representation used to detect clones. A token is a group of characters that form an atomic element in a string.

Twins role: occurs when the first cloned fragment that a method has is created because it is copied in all the methods of the family. It is detected by analyzing the changes in the methods that become part of the clone family when the clone is created. If all fragments that became cloned are changed when the clone relation is created, it is because the cloned fragment was introduced to all methods at the same time, in this case all cloned fragments are identified as twins.

Violation of design heuristics: is an implementation that does not comply with implementation heuristics proposed to conform to design principles.

Was cloned: is a characteristic of the relation between method and its clones It refers to the fact of having had clones; it is defined as 1 if the method had cloned fragments at any point in its lifetime, and 0 otherwise.